Algorithmic Skeletons for Exact Combinatorial Search at Scale

Blair Archibald

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

November 4, 2018

Abstract

Exact combinatorial search is essential to a wide range of application areas including constraint optimisation, graph matching, and computer algebra. Solutions to combinatorial problems are found by systematically exploring a search space, either to enumerate solutions, determine if a specific solution exists, or to find an optimal solution. Combinatorial searches are computationally hard both in theory and practice, and efficiently exploring the huge number of combinations is a real challenge, often addressed using approximate search algorithms. Alternatively, exact search can be parallelised to reduce execution time. However, parallel search is challenging due to both highly irregular search trees and sensitivity to search order, leading to anomalies that can cause unexpected speedups and slowdowns. As core counts continue to grow, parallel search becomes increasingly useful for improving the performance of existing searches, and allowing larger instances to be solved.

A high-level approach to parallel search allows non-expert users to benefit from increasing core counts. Algorithmic Skeletons provide reusable implementations of common parallelism patterns that are parameterised with user code which determines the specific computation, e.g. a particular search. We define a set of skeletons for exact search, requiring the user to provide in the minimal case a single class that specifies how the search tree is generated and a parameter that specifies the type of search required. The five are: Sequential search; three general-purpose parallel search methods: Depth-Bounded, Stack-Stealing, and Budget; and a specific parallel search method, Ordered, that guarantees replicable performance. We implement and evaluate the skeletons in a new C++ parallel search framework, YewPar. YewPar provides both high-level skeletons and low-level search specific schedulers and utilities to deal with the irregularity of search and knowledge exchange between workers. YewPar is based on the HPX library for distributed task-parallelism potentially allowing search to execute on multi-cores, clusters, cloud, and high performance computing systems.

Underpinning the skeleton design is a novel formal model, MT^3, a parallel operational semantics that describes multi-threaded tree traversals, allowing reasoning about parallel search, e.g. describing common parallel search phenomena such as performance anomalies.

YewPar is evaluated using seven different search applications (and over 25 specific instances): Maximum Clique, k-Clique, Subgraph Isomorphism, Travelling Salesperson, Binary Knapsack, Enumerating Numerical Semigroups, and the Unbalanced Tree Search Benchmark. The search instances are evaluated at multiple scales from 1 to 255 workers, on a 17 host, 272 core Beowulf cluster. The overheads of the skeletons are low, with a mean 6.1% slowdown compared to hand-coded sequential implementation. Crucially, for all search applications YewPar reduces search times by an order of magnitude, i.e hours/minutes to minutes/seconds, and we commonly see greater than 60% (average) parallel efficiency speedups for up to 255 workers. Comparing skeleton performance reveals that no one skeleton is best for all

searches, highlighting a benefit of a skeleton approach that allows multiple parallelisations to be explored with minimal refactoring.

The Ordered skeleton avoids slowdown anomalies where, due to search knowledge being order dependent, a parallel search takes longer than a sequential search. Analysis of Ordered shows that, while being 41% slower on average (73% worse-case) than Depth-Bounded, in nearly all cases it maintains the following replicable performance properties: 1) parallel executions are no slower than one worker sequential executions 2) runtimes do not increase as workers are added, and 3) variance between repeated runs is low. In particular, where Ordered maintains a relative standard deviation (RSD) of less than 15%, Depth-Bounded suffers from an RSD greater than 50%, showing the importance of carefully controlling search orders for repeatability.

Acknowledgements

Utmost thanks goes to Phil Trinder, Patrick Maier, and Rob Stewart, for their unfaltering encouragement, guidance and ideas. All have been simultaneously supervisors, colleagues, mentors and friends. This thesis would not have been possible without their support.

Thank you to everyone in the School of Computing science at Glasgow, my home for the last seven years. Special thanks to Jeremy Singer, for kindling my interest in research, and Ciaran McCreesh for sending me down the rabbit-hole of parallel search problems and for feedback on this thesis. Also to Stephen McQuistin for feedback on an earlier draft.

To all those at EPCC, especially Michele Weiland and Nick Johnson, thank you for hosting me during my internship. I learned a great deal about high performance computing and, of course, cryptic crosswords!

Thanks to the Software Sustainability Institute who taught me the wider importance of computing and ensured I didn't miss the forest for the (search) trees.

I am especially grateful to my parents, family, friends, and Jena who have formed the best support network I could ask for. Thank you.

Finally, thanks to Chris Jefferson, David Manlove, and Michel Steuwer for making the viva an enjoyable experience.

This work was funded from EPSRC grants: EP/K503058/1, EP/L50497X/1 and EP/M508056/1.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Exact combinatorial search algorithms are used in a wide range of application areas including constraint programming, computational algebra and many more [1]. Combinatorial searches look for configurations of domain-specific objects solutions that satisfy a set of goals. For example, we may wish to find cliques within graphs, i.e. a set of vertices such that each vertex is adjacent to every other vertex in the set. The specific configurations required are defined by the type of search: enumeration, which searches for all solutions matching some property, e.g. maximal cliques; decision, which looks for a specific solution, e.g. a clique of size k; or optimisation, which looks for a solution that minimises/maximises an objective function, e.g. finding a maximum clique. These problems are solved by systematic exploration of all valid object configurations known as the search space. For decision and optimisation problems the entire search space must be explored, or pruned, while decision problems may terminate early if a solution is found.

Generally a huge number of combinations exists, many that are no good, making these problems computationally hard in both theory (often $\mathcal{NP}$-hard [2]) and in practice, with large instances taking many hours, days, or more, to solve. Handling the large number of combinations is a real challenge, often dealt with by providing an approximate solution from (meta-)heuristic search algorithms [3]. However for many applications an exact or optimal solution must be guaranteed, for example, it is of little use to know it is *unlikely* a sub-structure exists for a mathematical object, we must be sure.

An alternative to finding approximate solutions is to exploit parallelism to reduce search times. At the core of exact search is a backtracking search algorithm that allows the search space to be represented as a tree. By providing parallel tree search functionality, exact combinatorial search problems may be parallelised in a domain-independent manner.

Modern hardware offers parallelism at every level, from multi-core parallelism on a single machine, through small clusters of machines, and up to large scale distributed cloud services and high-performance computing environments. The potential for parallelism in combinatorial

search is great, with many areas of the search space able to be explored with (almost) complete independence. However, despite the potential for speedups from exploiting parallelism, implementation challenges mean that most exact combinatorial search algorithms are either not parallelised at all, or are parallelised once using ad-hoc methods aimed at a single scale of parallelism and a specific application.

There is potential for a unified approach to exact parallel combinatorial search that works at every scale (from desktop to large cluster, cloud, or HPC), while removing the burden of parallel programming from the search domain expert. This is not simple: combinatorial searches differ from standard parallel workloads due to the speculative nature of the parallelism, their high degree of irregularity, heavy use of symbolic/integer methods as opposed to floating point, and minimal I/O.

This thesis presents the design and implementation of a set of parallel algorithmic skeletons for combinatorial search as a step towards this unified approach. The skeletons are general-purpose, allowing many different search applications to be encoded, while also allowing different parallelism approaches to be explored. By supporting distributed-memory architectures they provide easy access to the plethora of parallel hardware without requiring detailed parallelism knowledge from domain experts.

We show the approach to be general, encompassing many types of search (enumeration, decision, and optimisation) and many different applications, while also being scalable on medium sized clusters featuring 255 workers. By making it easier for domain experts to exploit parallelism, we allow them to solve larger instances, making previously impractical searches practical.

1.1 Contributions

This thesis makes the following research contributions:

1. **A critical review of parallelism for *exact* combinatorial search** (Chapter 2). Existing approaches for exact parallel combinatorial search often lack generality, i.e. they focus on a specific type of search such as optimisation, a specific application such as clique search, or a specific domain such as constraint programming. This thesis considers the differences and similarities across application domains, search problems, and search types, to inform the design choices for constructing a general-purpose and widely applicable parallel framework for exact combinatorial search. We explore and categorise existing approaches to parallel search (Section 2.4), including their scalability (Section 2.4.3), and use this to inform the design of algorithmic skeletons for search.

While there are many existing parallel search approaches we motivate the need for a new framework in Section 2.4.4.

2. **A novel formal model, MT^3, for parallel backtracking search, covering enumeration, decision, and optimisation search types** (Chapter 3). The formal model is based on operational semantics, and provides a precise specification of parallel search order and parallel reduction rules. The model is used to derive a programming interface for the skeletons (Section 3.4), inform the design of an abstract framework (TSF) for describing the skeleton operation (Section 4.2), succinctly describe the work generation behaviours of the general-purpose skeletons (Sections 4.3.4.1, 4.3.5.1 and 4.3.6.1), and to show how performance anomalies affect parallel search (Section 7.2).

3. **A widely applicable application programming interface for tree search** (Section 3.4). We introduce *Lazy Node Generators* as a uniform abstraction for application developers to specify how specific search trees are created. The generators implicitly encode application-specific search order heuristics. Search tree nodes are constructed lazily, such that pruning eliminates redundant computation, i.e. it eliminates sub-trees before they manifest. The generality of the Lazy Node Generator programming interface is shown by specifying the search trees of seven search applications (Section 5.2).

4. **The design and implementation of four, widely applicable, parallel algorithmic skeletons for tree search** (Chapter 4). We abstract parallel search into a family of algorithmic skeletons that use the Lazy Node Generator interface. Four skeletons are designed: *Sequential*, *Depth-Bounded*, *Stack Stealing* and *Budget*; based on parallel search approaches identified in Chapter 2. They are reusable, widely applicable, and reduce the engineering effort required to create custom search parallelisations. By widely applicable we mean the implementations are general enough to express enumeration, decision, and optimisation search types. Reusability is demonstrated by applying the skeletons to seven search applications.

5. **The design and implementation of a general-purpose parallel tree search framework: YewPar** (Chapter 5). YewPar is a C++ framework that implements the Lazy Node Generator programming interface (Section 3.4) and parallel search skeletons (Chapter 4) for distributed-memory architectures. It features custom work-stealing schedulers that handle search irregularity and carefully manage search order heuristics. YewPar builds on the HPX C++ parallelism framework to provide distributed-memory parallelism features such as a global address spaces. Modern C++ implementation techniques, such as template metaprogramming, keep the overheads of YewPar small compared to state of the art search implementations, showing, on average, a 6.1% slowdown (Section 6.3).

6. **A systematic performance analysis of the skeletons and YewPar** (Chapter 6). A detailed analysis of the skeletons is performed using seven applications and more than 25 instances, including a mix of enumeration, decision, and optimisation searches. The applications are: *Counting Numerical Semigroups*, *Unbalanced Tree Search*, k*-Clique* to solve a finite geometry case study, *Subgraph Isomorphism Problem*, *Maximum Clique*, *Travelling Salesperson* and *0/1 Knapsack*. We find low overheads, with a mean 6.1% slowdown for using the skeletons as opposed to hand written searches (Section 6.3), and that knowledge exchange using a global address space and broadcasts are appropriate for medium scale search, e.g. 255 workers (Section 6.4). The skeletons reduce the runtimes of search by an order of magnitude, i.e. from hours/minutes to minutes/seconds. No single skeleton performs best for all applications (Section 6.8). Averaging over all applications, Stack-Stealing performs best with an average speedup of 37.4 on 120 workers, compared with 24 and 34 for Depth-Bounded and Budget respectively. The skeletons show good scalability on a set of larger instances (Section 6.9) achieving a minimum (geometric mean) 45% efficiency and with a maximum 112%[1] efficiency on 255 workers.

7. **The design and implementation of a specialised skeleton for replicable branch and bound search** (Chapter 7). Search order anomalies in parallel branch and bound search can cause huge variance of runtimes, even for the same instance. Anomalies occur when a parallel run performs significantly more (or less) work than a sequential search due to runtime knowledge sharing and a dynamically changing search tree shape. A lack of reproducible runtimes is particularly undesirable for domains such as empirical algorithmic design. We show the design and implementation of a specialised anomaly-avoiding skeleton, *Ordered*, that attempts to provide the following properties **a)** parallel runs are never slower than the single worker case; **b)** adding additional workers does not cause significant degradation of runtime (avoid slowdown anomalies) and **c)** runtime variance is small. Analysis of Ordered shows that, while being 41% slower on average (72% worse-case) than Depth-Bounded[2], in all cases it maintains the replicable performance properties (allowing for small slowdowns due to parallel overheads). In particular, where Ordered can maintain a relative standard deviation (RSD) of less than 15% in all cases, Depth-Bounded suffers from an RSD greater than 50%, showing the importance of carefully controlling search orders for repeatability.

[1] Efficiencies >100% are possible due to superlinear scaling behaviour.
[2] For Maximum Clique.

1.2 Publications and Authorship

This thesis is closely related to the work reported in the following publications:

- B Archibald, P Maier, C McCreesh, R Stewart and P Trinder, **Replicable Parallel Branch and Bound Search**, Journal of Parallel and Distributed Computing, Volume 113, 2018, https://doi.org/10.1016/j.jpdc.2017.10.010. (reference [4]).
- B Archibald, P Maier, R Stewart, P Trinder, and J De Beule. **Towards Generic Scalable Parallel Combinatorial Search**, in Proceedings of the International Workshop on Parallel Symbolic Computation, 2017, https://doi.org/10.1145/3115936.3115942. (reference [5]).

The work reported here is primarily my own, with the following exceptions:

- The definitions of search trees and initial operational semantics for parallel tree traversal (i.e. those reported in Archibald et al. [4]) in Chapter 3 are the work of Patrick Maier. Changes to these rules to support specific search types and spawning is my own work.
- Background information for the finite geometry case study (Section 5.2.2.2) is based on a description written by Jan De Buele (reported in Archibald et al. [5]).
- Sequential implementations for clique search (Section 5.2.2.1, Section 5.2.3.1) and subgraph isomorphism (Section 5.2.2.3) were provided by Ciaran McCreesh [6].
- Sequential implementation for Numerical Semigroups (Section 5.2.1.2) was provided by Florent Hivert [7].
- The formula for repeatability of scaling in Appendix C is based on a derivation from Patrick Maier.

Chapter 2

Background

This thesis is concerned with parallel algorithmic skeletons applied to exact combinatorial search. This chapter introduces the technical background of both combinatorial search and parallelism separately, before combining them to detail the key challenges of parallel combinatorial search and discuss existing approaches.

Combinatorial search is introduced with an example, namely finding cliques within graphs (Section 2.1), before being generalised to wider classes of application (Section 2.1.1). Combinatorial search solves a family of problems, based on a specific search type: enumeration, decision or optimisation searches (described in Section 2.1.3).

Parallelism can help handle the computational complexity of search, allowing us to solve problems that would otherwise be impractical. A general discussion of parallelism is given in Section 2.2. To allow scalability we focus on distributed parallel hardware (Section 2.2.1), such as that found in clusters, cloud and high performance computing (HPC) environments. Background on parallel programming models is given (Section 2.2.2), with particular focus on task-parallelism and the related problem of (distributed) load-balancing. We finish this section with a discussion of high-level approaches to parallelism (Section 2.2.4), in particular parallel algorithmic skeletons.

Parallel exact combinatorial search is discussed in Section 2.3. Search can be parallelised in many ways, e.g. parallel node processing, space-splitting, or portfolios (Section 2.3.1). We focus on space-splitting as it is domain-independent, making it an ideal candidate for general-purpose skeletons. The key challenges of parallel combinatorial search are introduced (Section 2.3.2), in particular highly irregular task sizes and performance anomalies caused by ordering effects.

Existing approaches to space-splitting parallel search are categorised and discussed in Section 2.4. These approaches influence the skeleton designs of Chapter 4. Although there are many existing approaches, we argue in Section 2.4.4 that a new parallel search framework–that

unifies many of these approaches–is required.

2.1 Combinatorial Search

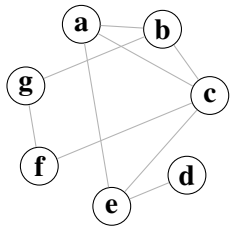


Figure 2.1: An example graph.

We begin by considering an example combinatorial search application: finding cliques within a graph such as the one shown in Figure 2.1. In practice the graphs are much larger, often with hundreds or more vertices.

A clique is a set of vertices C where each vertex in C is adjacent to every other vertex in C. Example cliques in Figure 2.1 are $\{a\}$, $\{a, b\}$ and $\{a, c, e\}$. $\{a, b, c, e\}$ is not a clique as there is no edge between e and b.

We are interested in particular properties of cliques within graphs. We may wish to know, for example, the number of maximal cliques[1]: which is of practical importance, e.g. for bioinformatics applications [8]. Or perhaps we want to determine if a clique of size k is in the graph; the k-clique problem (described in Section 5.2.2.1). Finally, we may look for the largest clique(s) in the graph; the Maximum Clique problem (described in Section 5.2.3.1). In each case we are looking for specific *combinations* of vertices from within the larger graph structure.

A naive approach to clique search would be to enumerate the powerset of all vertices (i.e. $\{\{\}, \{a\}, \ldots, \{a, b, c, d, e, f, g\}\}$) and then, in turn, test each to check if it maintains the clique property. The largest element(s) of the powerset meeting the clique poperty would be the maximum clique(s); k-cliques are any powerset elements with cardinality k that meet the clique property; and maximal cliques are those that meet the clique property and are not a proper subset of any other set that also meets the clique property. While such an approach can be applied to small graphs, the size of the powerset grows exponentially with the number of vertices in the graph, making this approach computationally impractical for larger graphs.

Rather than generating all possible combinations of vertices, we can search more efficiently by ensuring we only *generate* valid cliques. That is, given any clique, we can extend it by adding any vertex that is adjacent to all elements of the clique. This is the essence of backtracking search, where we continually extend a clique until no more additions are possible. When this occurs we remove the last added element and try to add a different vertex. We can continue to add and remove vertices systematically, building a search space of valid cliques. This search space may be represented as a tree such as in Figure 2.2(a), where nodes in the search tree

[1] A maximal clique is a clique where it is not possible to add any additional vertices without breaking the clique property. For example, $\{a, b\}$ is not maximal as we can add c and still form a clique, while $\{d, e\}$ is maximal as we cannot add any more vertices without breaking the clique property.

represent cliques. Notice that some cliques are generated multiple times as $\{a, b\} = \{b, a\}$. We can avoid this by only generating cliques we have not seen before as in Figure 2.2(b). In practice this tree is never fully materialised in memory, and search creates nodes on demand when they are required.

We can explore the search tree to find the information we require. The number of maximal cliques is the number of (unique) leaf nodes in Figure 2.2(a), 6 in this case. We must use the tree that does not remove symmetries here since a leaf node in Figure 2.2(b) is not guaranteed to be maximal (e.g. adding a to the leaf node $\{b, c\}$ still forms a clique). In practice the Bron-Kerbosch algorithm [9] is used to find maximal cliques and ensures we build a backtracking search tree without symmetries, with maximal cliques at the leaf nodes. The maximum clique is the longest branch(es) of Figure 2.2(b). In this case $\{a, b, c\}$ and $\{a, c, e\}$ are both maximum cliques. k-cliques are nodes at a given depth of Figure 2.2(b), e.g. 2-cliques are at depth 2. There are 9 2-cliques, but no 4-cliques.

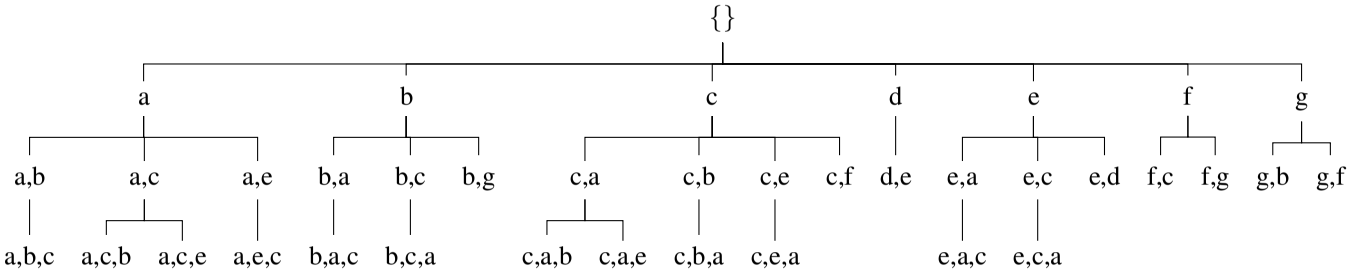
It is not always necessary to traverse the entire search tree. For example, to prove a k-clique exists it is sufficient to find a single example, i.e. we only need to generate $\{a\}$ and $\{a, b\}$ to show a 2-clique exists. To show a k-clique does not exist then we must search the entire space to prove that no example exists. To enumerate all maximal cliques we must explore the entire search tree. For finding a maximum clique we must first find the optimally sized clique and then search the rest of the space to prove that there is no larger clique.

In practice, for k-clique and maximum clique, we do not need to explore all branches so long as we can prove a k-clique or improved optimal clique cannot exist in the branch. This is the basis of branch and bound search. During the search the best solution found so far, known as the incumbent, is tracked. A bounding function is then applied to each node that estimates the maximum possible improvement if we continue searching from this node. If the bound is less that our k value, or current optimal solution, then we can safely ignore it without affecting the proof of existence or optimality. For clique search, a possible bounding algorithm is greedy colouring [10], where the number of colours used bounds the maximum size of a clique.

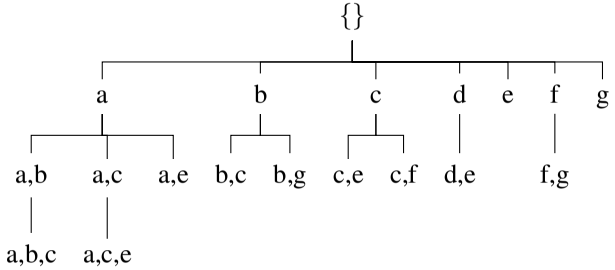
2.1.1 General Combinatorial Search

While we have used clique search as an example, the concepts apply in the more general context of combinatorial search. Combinatorial search looks for specific configurations of objects, e.g. cliques, within the set of all object combinations (the search space), e.g. the powerset of vertices. Search spaces are often huge and search problems are hard in both theory, usually $\mathcal{NP}$-hard [2, 11], and in practice, with instances often taking hours, or more, to solve.

Combinatorial problems are typically solved using either:



(a) With symmetries.



(b) Without symmetries.

Figure 2.2: Search tree for clique search of Figure 2.1

Listing 2.1: General Backtracking Search.

```
1  function search(SearchSpace space, Node root):
2    expand(space, root)
3
4  function expand(SearchSpace space, Node n):
5
6    // Searches perform processing of 'n' here. The type of processing depends on
7    // what is being searched for, for example, checking if 'n' is a valid solution.
8    // Node processing can stop the next branching step being executed to allow support
9    // for pruning and early termination.
10
11   children ← branch(space,n)
12   for c in children:
13     expand(space, c)
14   return // Backtrack when no children remain
```

Approximate Methods that trade improved performance at the cost of non-exact answers, i.e. they do not *guarantee* that the entire search space is explored. Example approximate methods are simulated annealing and ant colony optimisation [3]. In the context of clique search, they are not appropriate for maximal clique enumeration as maximal cliques may be missed, for k-clique they can only solve satisfiable instances but cannot prove that an instance is unsatisfiable, and for maximum clique they cannot guarantee optimality.

Exact Methods that ensure the entire search space is systematically explored, usually by some form of backtracking search, allowing *proofs* of optimality etc. to be obtained. Exact searches may be converted to an approximation algorithm by stopping search once a suitable timeout has elapsed.

This thesis attempts to achieve the benefits of exact search, while still allowing larger instances to be solved by using parallelism to reduce search runtime rather than approximate methods.

2.1.2 Backtracking Search

Exact search algorithms work by systematically generating (valid) object configurations and testing them for specific properties. Most searches[2] are based around a backtracking search algorithm such as that of Listing 2.1. At each step a `branch` function (line 5) determines all valid children of a node n, e.g. a clique (represented by n) extended with additional vertices. These children are explored in turn until no children remain. Once all children are explored we backtrack and try the next child at the previous level of search. This ensures no part of the search space is unexplored.

[2]Some search algorithms, such as the conflict driven clause learning (CDCL) for SAT problems [12], use search restarts and non-chronological backtracking when performing search. We do not consider these types of algorithms in this thesis.

2.1.3 Types of Search

The backtracking search algorithm (Listing 2.1) includes node processing that is dependent on the type of search. There are three main types of search:

Enumeration: Enumeration searches determine properties of the search space, for example counting maximal cliques in a graph (Section 2.1). At each expand step, node processing first determines if we are interested in the node, e.g. "is it maximal?", and reacts accordingly, e.g. incrementing a counter or storing a representation of the node. Enumeration problems must explore the full search space to ensure correctness, giving them a fixed workload. Examples of enumeration searches include the well-known n-queens problem, where we wish to enumerate ways of placing n queens on a $n \times n$ chess board such that they do not attack each other [13]; enumerating all maximal cliques of a graph [8]; and counting numerical semigroups of genus g (Section 5.2.1.2).

Decision: Decision searches look for a specific node that matches a property, for example searching for a k-clique in a graph (Section 2.1). If a node with the property exists we call the instance *satisfiable* otherwise it is *unsatisfiable*.

Decision searches only need to explore enough of the search space to determine if a solution exists, allowing early termination for satisfiable instances. If no solution exists then the entire space must be explored to prove no solution exists. Examples of decision searches are the k-clique problem (Section 5.2.2.1); subgraph isomorphism finding (Section 5.2.2.3); and the boolean satisfiability problem (SAT) [14] that determines a valid variable assignment in a boolean formula that allows it to be true (if one exists).

Optimisation: Optimisation searches look for node(s) that maximise or minimise a given objective, for example, finding the largest clique in a graph (Section 2.1). The entire search space must be explored (or eliminated by a bound) to first find the optimal solution and then to prove that no better solution exists. Examples of optimisation searches are finding the maximum clique in a graph (Section 5.2.3.1); optimally packing items into a Knapsack (Section 5.2.3.3); and finding the lowest cost tour in the travelling salesperson problem (Section 5.2.3.2).

2.1.4 Search Ordering and Heuristics

Finding a solution early allows early termination in decision searches and improved pruning for branch and bound optimisation searches. To exploit this fact the order in which the search tree is explored is extremely important. For enumeration problems the entire search space is explored so ordering has little overall effect, other than changing performance characteristics such as overall memory usage.

Listing 2.2: Backtracking Search using a selection function.

```
function search(SearchSpace space, Node root):
  frontier ← {root}
  expand(space, frontier)

function expand(SearchSpace space, NodeSet frontier):
  forever:
    n ← select(frontier)
    if n == Nothing:
      return // Search complete
    else:

      // Searches perform processing of 'n' here. The type of processing depends on
      // what is being searched for, for example, checking if 'n' is a valid solution.
      // Node processing can stop the next branching step being executed to allow support
      // for pruning and early termination.

      children ← branch(space,n)
      frontier.insert(children)
```

The backtracking search algorithm given in Listing 2.1 implicitly explores the search tree in a depth-first manner, where the deepest unexplored node is expanded; usually tie-breaking by taking the leftmost node first if there are two unexplored nodes at the same depth. In practice different node selection policies are available, and are described using a *selection* or *heuristic function* [15]. These functions operate on a set of unexpanded nodes (the search frontier), as shown in Listing 2.2.

Three common search orderings are:

Depth-First Search (DFS): chooses the first open child at the highest depth (deepest in the tree). The main benefit of DFS is modest memory requirements, as we only need to store the path from the root to the current node (and siblings on the path) and once nodes are processed they can be safely removed from memory. In practice sibling nodes may be generated on-demand, e.g. created in the for loop of Listing 2.1, reducing the memory requirement further.

However, DFS has a narrow view of search, i.e. most nodes share common ancestors, and can spend a long time exploring a single branch. The use of heuristics to improve branch selection for DFS is discussed in Section 2.1.4.1.

Breadth First Search (BFS): chooses the first open child at the lowest depth in the tree. While this gives it a wide view of search, it suffers from large memory requirements to store all nodes (required to expand all children node beneath them). This large memory requirement often excludes BFS from search problems due to the huge number of nodes in a search tree. Practically it is often useful to use BFS to generate initial tasks for a parallel search (Section 2.4.1).

Best First Search (BeFS): given a suitable bounding function, chooses an open child with the best bound, i.e. the one most likely to lead to an improved solution. The performance

of BeFS depends on how good the bounding function is. If many nodes are assigned the same bound then it can lead to an increase in memory requirements similar to BFS. On the other hand, given a strong bounding function, search can quickly converge on a solution.

The skeletons presented this thesis primarily use DFS, as it (a) runs in bounded memory, a limiting factor in parallel search and (b) does not require a global ordering on nodes as in BeFS.

2.1.4.1 Search Heuristics and Discrepancies

DFS can spend a long time exploring a single sub-tree. If search chooses a poor sub-tree to explore, i.e. one with limited solutions, then it can be a while before this decision can be remedied by choosing a better sub-tree.

Search heuristics attempt to avoid this issue by placing an ordering on how the nodes are explored. In general, for DFS, these manifest as a left-to-right ordering on the children of a node, where nodes to the left are heuristically good choices. For example, when solving a Travelling Salesperson Problem children may be ordered in increasing distance cost from the last city added to the tour.

Domains such as constraint programming [16], that try to find a set of assignments of the form $x = a$, often include two types of heuristic: variable ordering and value ordering. Variable orders determine the next variable, e.g. x, to be assigned, while value orderings determine the value, e.g. a. As such, variable orders control the tree shape vertically and value orders horizontally (where branching represents assignment of a value to a variable). Heuristics determine good choices, not necessarily those that lead to solutions. For example, a common variable ordering heuristic is fail-first [17] that can reduce the average depth of tree branches by backtracking sooner (i.e. on failing to find a valid assignment).

In this thesis, unless otherwise stated, we always use heuristic ordering to mean the left-to-right ordering on child nodes. This does not preclude variable ordering heuristics, only that these are implicit in the branching step. Given the importance of search heuristics, any approach to general-purpose parallel search must be able to encode domain-specific heuristics.

Unfortunately heuristics are often weak near the top of the search, where there is little information available to fully inform them [18]. This can lead to situations where DFS gets stuck exploring a poor sub-tree due to a bad heuristic choice near the start of search.

Discrepancy search [18, 19] is one method for avoiding this situation. A discrepancy occurs when search goes against the heuristic ordering, i.e. it takes a right child instead of a left child. Discrepancy searches purposefully use discrepancies to counteract poor early decisions, by

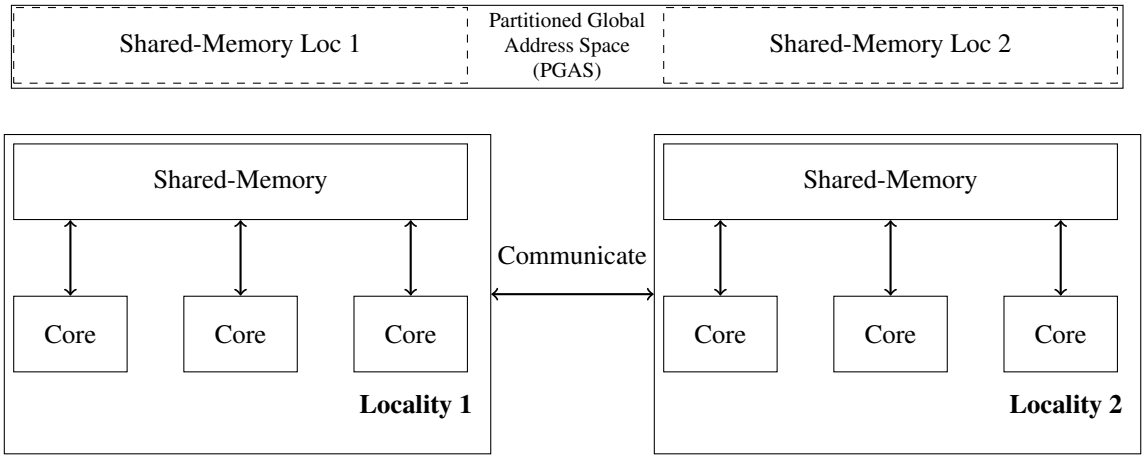


Figure 2.3: Example distributed parallel architecture.

iteratively probing the tree based on the number of discrepancies. That is, it explores branches with no discrepancies, then those with one discrepancy, then two, and so on.

We use discrepancy search orders in Section 7.4.1 as a method of avoiding bad heuristic choices in a fixed-order parallel search.

2.2 Parallelism

This section introduces key parallel computing concepts used in the thesis. Section 2.2.1 gives a high-level overview of parallel architectures, in particular those that achieve scalability by distributing computation over multiple cooperating machines. Section 2.2.2 discusses two main models for (user-level) parallelism: data-parallelism and task-parallelism. Section 2.2.3 looks at techniques for mapping computations to workers, in particular the core work-stealing scheduling algorithm used in the tree search skeletons of Chapter 4. Finally an overview of the typical parallel performance metrics is provided (Section 2.2.5).

2.2.1 Parallel Architectures

For many years programmers could rely on exponentially increasing processor clock speeds for improved single-processor performance without requiring structural changes to their applications: the so-called “free lunch” [20]. Due to hardware challenges such as heat dissipation and power consumption, processor clock speeds stopped significantly increasing around 2003. To further improve performance, modern processors consist of multiple cooperating processors/processor cores (multi-core computing). While additional cores enable concurrent execution of operating system processes, to take full advantage of extra cores for specific applications, large and often difficult to correctly implement, application changes are required.

Figure 2.3 shows an example of a modern distributed parallel architecture that consists of multiple physical machines, called localities[3], connected via a network. While only two localities are shown, in practice many more localities are available.

Localities feature multiple processor cores (and sometimes multiple processors) that share a single address space, the shared-memory model. This shared address space allows for quick (compared to distributed) communication between cores. We assume that localities are formed of a set of parallel cores without considering specific architectural features, i.e. the layout of processors and caches. The term *worker* is used in a general sense to refer to a processing element, e.g. a core, a thread, or even a process in distributed single core architectures.

The number of cores within a single locality is limited by power requirements and heat dissipation. Larger parallel architectures are created using multiple cooperating localities, often communicating using message passing technologies such as MPI [21]. Distributed parallel systems of this kind are common in cluster, cloud, and high performance computing (HPC) setups. Usually localities are fully interconnected, allowing any locality to communicate with any other locality, although communication time is not guaranteed to be uniform, i.e. the underlying network topology may not be fully interconnected and messages may require multiple hops.

In a distributed architecture each locality has its own memory space (distributed-memory), requiring explicit communication to read/write data between localities. To improve programmability, the partitioned address space model (PGAS) provides a shared-memory abstraction for distributed-memory environments, allowing one locality to directly access the memory of the another locality, with the PGAS implicitly managing communication and address resolution. The YewPar implementation, described in Section 5.1.2, makes heavy use of the PGAS model to provide global knowledge transfer and load-balancing.

This thesis only considers homogeneous architectures where the performance of each locality is uniform.

2.2.2 Parallelism Models

Parallelism can be introduced to a system in many ways, from instruction level parallelism [22], such as superscalar processing, to hand-crafted multi-threaded applications.

User defined parallelism is often classified as either data-parallel or task-parallel, which represent approaches to structuring parallelism rather than specific implementation techniques. These categories are not mutually exclusive and a task-parallel application may use elements of data-parallelism, e.g. vector instructions.

[3]Localities are also commonly known as nodes. To avoid confusion with search tree nodes, we adopt the term localities here.

2.2.2.1 Data-Parallelism

Data-parallel applications divide data structures into parts and process each of these in parallel. For example, say we want to add two large arrays. In a sequential environment we would loop over both arrays, one element at a time, and perform the addition. A data-parallel approach would instead break the array into n chunks and distribute these across workers, each performing the addition for the assigned chunk.

Data-parallelism is exploited by many modern processors through vector instructions that allow, for example, 4 floating point numbers to be added using a single instruction [23]. This fact is often utilised by compiler designers to implicitly parallelise user code.

Data-parallelism has also gained popularity for processing large amounts of data in cloud computing environments. Data Driven Execution Engines, such as MapReduce [24] or Apache Spark [25], split large datasets, e.g. database tables, over multiple localities for processing, while transparently performing communication, load-balancing, and fault tolerance.

2.2.2.2 Task-Parallelism

Task-parallel applications are structured as multiple cooperating tasks, where a task is a sequence of instructions, e.g. a function. Tasks may have ordering dependencies, e.g. task A *runs after* task B, and, in many systems, tasks are allowed to generate new tasks.

Task-parallel frameworks/runtimes, for example Cilk++ [26], Thread Building Blocks (TBB) [27] or HPX [28], often implicitly handle task scheduling, allowing the programmer to focus on structuring the application correctly. Because of *runs after* relationships, applications should create more tasks than workers. Having more tasks than workers allows workers to make progress on other tasks instead of idling while awaiting data. The mapping of tasks to workers is discussed further in Section 2.2.3.

Task-parallel frameworks are formed, at a minimum, of a *spawn* operator that creates new tasks (often from a function), and a method to express task dependencies. Task dependencies are often expressed using parallel futures [29]. Futures represent the results of parallel computations, and may be either valid, if the parallel task has already executed, or awaiting a result. Tasks may wait for futures to become valid to express *runs after* relationships.

Spawning and task dependencies are shown for an example task-parallel Fibonacci program in Listing 2.3[4]. In this example a spawn primative converts a function (and any arguments) into a task and returns a future representing the result of the task. Importantly the execution of the function is delayed until the task is scheduled[5]. The `await` function blocks until all

[4]Expressed as pseudocode rather than using a specific parallel framework.

[5]This is in contrast to strict function calls where f(g(x)) initially evaluates g(x) and then invokes f with the result.

Listing 2.3: Fibonacci as a task-parallel program.

```
function fib(n):
  // Trivial Case
  if n <= 1:
    return 1

  // Two tasks for the left and right Fibonacci tree branches
  future n_minus_one ← spawn(fib(n-1))
  future n_minus_two ← spawn(fib(n-2))

  // Block until results from both tasks are valid
  await(n_minus_one, n_minus_two)

  // Combine the results
  return n_minus_one.value + n_minus_two.value
```

futures passed to it contain a value, and is common in many task parallel frameworks.

As search trees are not fully materialised in memory, task-parallelism is the dominant parallelism model for tree search.

2.2.3 Load-Balancing

A challenging aspect of task-parallelism is ensuring workers are not under-utilised. A load-balancing scheme [30] provides an (often dynamic) assignment of tasks to workers. A system is described as starved if there are not enough tasks to keep workers busy, as often occurs at the beginning and end of application runs.

If tasks are known *a priori* then load-balancing can be performed statically [31]. Assuming no dependencies, tasks can be assigned to processing elements such that the average load of each is similar, avoiding the case where a single task dominates the total running time.

Unfortunately static schemes cannot handle dynamic and irregular computations. Dynamic in the sense that tasks may generate new tasks at runtime, and irregular due to unpredictable task runtimes. Tree search falls into this category of irregular application, where the running time of tasks is both unknown and varies significantly, e.g. minutes versus milliseconds.

To handle irregular applications, *dynamic* load-balancing, which can respond at runtime to load imbalance, is required. Dynamic load-balancing schemes fall into one of two categories, based on where the re-balancing decisions are made:

Centralised A master process collates information from all processing elements and uses this to make informed load-balancing decisions, e.g. [32]. The global view of system state allows precise load rebalancing to be performed at the cost of increased communication to a single locality, which may become a bottleneck for large systems. Some centralised schemes operate in a hierarchical manner, where load balance within smaller clusters is maintained using a single master per cluster, and these masters then communicate to

make global load-balancing decisions. These hierarchical setups are typically known as master–hub–worker schemes.

Decentralised In a decentralised approach the workers themselves are responsible for maintaining load balance. Work-stealing scheduling [33] is a famous decentralised approach that shines for irregular computations and is the form of load-balancing used in this work.

2.2.3.1 Work-Stealing

In a work-stealing scheme, tasks are divided up into multiple workpools that store ready-to-run tasks (those that are not waiting for other tasks to complete). The number of workpools may vary, for example there may be one workpool per worker, or one workpool per-locality in the case of distributed setups[6]. Each worker is assigned a local workpool, where newly-generated tasks from this worker are placed.

Workers indefinitely fetch and execute tasks from their local workpool so long as tasks are available. When the workpool is empty[7] the work-stealing algorithm is initiated to avoid the worker becoming idle (if possible).

The idle worker becomes a *thief* and selects another worker, the *victim*, to steal tasks from. The thief accesses the victims workpool (either directly or via a request to the victim) and, if it is non-empty, steals work from the victims workpool and adds it to its own. If no work is available at the victim[8] then the thief will choose a new victim and repeat the process.

In practice, work-stealing algorithms often differ on:

1. Victim selection, with random victim selection being popular due to strong theoretical performance guarantees [34].

2. How much to steal from a victims workpool: often this is a single task or a steal-half approach [35].

3. Whether steal requests are forwarded to other workers in the case of a failed steal.

As work-stealing makes no distinction about *where* the victim resides it is applicable both in shared-memory and distributed-memory environments.

[6]The skeleton implementations Section 5.1.3 use a workpool per-locality, shared between all workers.

[7]A scheme known as watermarking allows work-stealing to begin when the number of tasks in the workpool goes below some threshold, allowing communication to overlap with computation avoiding worker stalls.

[8]Sometimes the victim will forward the steal request to another worker instead of signalling the original thief.

The workpools are often based on double-ended queues (deque) that allow different behaviour for local and remote steal requests, e.g. local workers take from the front (newest tasks) and remote workers from the rear (oldest, and hopefully largest, tasks). We discuss the implications of deque based work-stealing for search applications in Section 4.4.

2.2.4 High-Level Approaches to Parallelism

Although task-parallel frameworks improve programmability by introducing features such as automatic scheduling, creating task-parallel applications still requires parallelism knowledge to correctly structure the application. For example, tasks must be created at a suitable granularity, task dependencies must be managed to ensure deadlock is not introduced, and updates to shared data-structures require careful management.

To make parallelism available to non-experts, higher level approaches are required.

Higher level approaches to parallelism target specific types of (recurring) parallel application. For example in OpenMP [36] a user can parallelise a for loop by using the `#omp parallel for` pragma[9] and have OpenMP automatically decompose the loop into chunks, schedule them for execution, and ensure threads re-join after the loop.

2.2.4.1 Algorithmic Skeletons

Algorithmic skeletons [37], introduced by Cole [38], provide a method to structure parallel programs as a set of higher order functions that abstract over common patterns of parallel coordination. The skeletons themselves are parameterised with user-specific computations allowing a single skeleton to be applied in multiple domains. For example, a parallel *map* skeleton applies the user-specified function to each element of a collection in parallel. The users task is greatly simplified as they need not be aware how the map is implemented, only of the semantic outcome of the operation. Because the semantics are well defined, larger applications can be constructed using multiple skeletons. Importantly, the user does not need to use any low-level parallel features, e.g. locks, directly, allowing issues such as deadlocks to be avoided.

By abstracting away details of the underlying coordination layer, algorithmic skeletons gain portability in their parallel implementations. For example a parallel map might use a GPU implementation [39], a shared-memory implementation utilising a framework such as OpenMP [36], or split the collection over a set of distributed machines.

Common examples of algorithmic skeletons include parallel *reduction*, which combines elements of a collection, e.g. summing an array; *pipelines*, which transform input data through

[9] Assuming loop iterations are independent, i.e. iteration i does not depend on iteration $i-1$ and there is no shared data. Some loops with shared data can be specified using additional pragmas.

a set of functions simultaneously executing each function on the output of the previous step; and *divide-and-conquer*, which splits the input into smaller tasks, executes a function on trivial elements, and recombines the elements to compute a final result.

A detailed list of skeletons is given by González-Vélez and Leyton [40]. González-Vélez and Leyton split skeletons into three categories: data-parallel, task-parallel, and resolution. Resolution skeletons abstract over families of problems, and as such, search skeletons fit into this category. That is, they are reusable for a range of parallel searches, but are not necessarily as general-purpose as, for example, a map.

Sometimes parallel implementation details necessarily leak into the skeletons: for example, a distributed-memory map over a non-primitive data type requires a method to serialise the type. Some skeletons give additional control of parallelism to the user by, for example, letting them fine tune parameters such as chunk sizes, which necessarily leaks the fact that chunking is being used.

2.2.5 Parallel Performance Metrics

The overall goal of parallelism is to perform computations faster than is possible using a single worker. The following metrics are used to discuss the performance of parallel applications.

Sequential Runtime T_{seq}: Total runtime for a *fully* sequential run on a single worker.

Parallel Runtime $T_{par(n)}$: Total runtime for a parallel run using n workers. Generally, $T_{par(1)} > T_{seq}$ as this includes parallelism overheads such as acquiring locks.

Absolute Speedup $\frac{T_{seq}}{T_{par(n)}}$: Measures how much faster a parallel run with n workers is compared to a fully sequential run. If the workload of an application is fixed, i.e. does not depend on n, then the ideal speedup for n workers is n. In practice, due to sequential parts of applications as well as the effects of shared hardware, e.g. caches, memory busses etc., it is less than n.

Relative Speedup $\frac{T_{par(m)}}{T_{par(n)}}$: Measures how much faster a parallel run with n workers is compared to a parallel run with m workers. This is useful when it is impractical to gather T_{seq} for an application, e.g. it may take days to run without parallelism.

Parallel Efficiency $\frac{Speedup}{Workers}$: Usually given as a percentage, measures how well the parallel resources are utilised. A 100% efficient program gives fully linear scaling. For applications with non-fixed workloads, efficiency may be >100%.

Reasoning about speedups in parallel search can by difficult as they often feature non-fixed workloads. Non-fixed workloads occur when the total amount of processing for n workers

is significantly different from $n + 1$ workers. This can be caused by, for example, pruning of the search tree causing the total number of nodes searched to differ in the n worker and $n + 1$ worker searches. In this case, speedups of $> n$ for n workers can be observed as the total work performed is reduced as opposed to workers being used to process a fixed amount of work faster. Such a speedup is called a superlinear speedup. Caveats when reasoning about parallel search performance are discussed further in Section 6.2.

2.3 Parallel Combinatorial Search

This section introduces parallelism methods and key challenges for parallel search. Section 2.4 details specific existing approaches found in the literature.

The aim of parallel search is to both solve existing instances faster and allow larger instances to be practically solved. To make search practical we wish to reduce search times by an order of magnitude, that is take instances that run for hours and solve them in minutes, or from days to hours. Being able to solving instances faster has significant impact for many applications: for example, given a vehicle routing problem we cannot spend two days to find an optimal schedule if the vehicles must be dispatched on the same day.

While parallelism can solve instances that would be impractical otherwise, there are always harder instances to be solved, e.g. larger mathematical structures to search. Given the $\mathcal{NP}$-hard nature of search, processor counts are unlikely to keep up with the computational demands. However, processor counts are continuing to rise, with many high performance computing (HPC) systems featuring more than 300,000 cores [41]. Taking advantage of these existing cores is essential for solving instances that are currently out of reach today.

2.3.1 Methods of Parallelising Combinatorial Search

Adopting the taxonomy of Gendron and Crainic [42], tree search applications may be parallelised in three main ways:

Parallel Node Processing (Type 1 [42]): Parallel node processing introduces parallelism to branching/bounding steps of search tree generation. For example, the bounding operations for the Flowshop Problem can be performed on GPUs [43, 44, 45], or we can make use of vector instructions (e.g. Section 5.2.2.1) on CPUs. Parallel node processing does not change how the tree is explored; instead it tries to reduce the time spent processing each node.

Space-Splitting (Type 2 [42]): Space-splitting techniques divide the search tree into multiple (non-overlapping) sub-trees that are searched using multiple workers. Each sub-tree

can be searched fully independently; however, knowledge, such as improved bounds, is often shared between workers to improve performance by pruning. Sharing of knowledge between workers can cause interesting performance effects as discussed in Section 2.3.2.1.

Portfolio (Type 3 [42]): Portfolio techniques run multiple *complete* searches for a specific application. Each search differs in some manner, often by adopting different heuristics or bounding methods. This approach reduces the cost of search choosing a poor heuristics/bounding method. As with space-splitting searches may run completely independently, or share new knowledge as it is gained.

Hybrid approaches are also possible: for example, we can use a space-splitting search that performs bounding operations on a GPU, or run multiple space-splitting searches at once as a portfolio. It is not clear how hybrid methods interact on resource constrained systems, e.g. parallel node processing and space-splitting may share the same workers. Apart from using accelerators to perform node processing, we are unaware of work in this direction.

We are mostly concerned with parallelism via space-splitting as it leads to reusable approaches across a range of applications. Both parallel node processing and portfolio approaches require domain-specific knowledge, e.g. to vary bounding functions, limiting the generality of the approach[10]. However, we are cautious not to exclude a user from using parallel node processing within in a general-purpose framework, and many of the case studies in Section 5.2 make use of data-parallelism (vectorisation) as a form of parallel node processing.

2.3.2 Challenges of Space-Splitting Search

Space-splitting approaches map well into the task-parallel model (Section 2.2.2), with tasks representing sub-tree search. The search space is divided such that sub-trees are non-overlapping, allowing search to be performed with no interaction between tasks in the case of enumeration searches, and with limited interaction in the form of knowledge exchange (e.g. new bounds) for branch and bound decision and optimisation searches. Task input is limited, requiring only the sub-tree root.

The (near) independence of tasks gives this approach a deceptively simple feel. However, there are many challenges to overcome. Search trees are highly irregular[11] and the time required to search a particular sub-tree is both unpredictable and highly variable.

[10] A skeleton for portfolio based searches is possible and, if knowledge exchange is excluded, resembles a task farm with early termination.

[11] See, for example, Figure 4 of [46]

The irregularly of the tasks can cause a single task to dominate the running time. Careful choice of an initial work distribution, or methods to dynamically generate more work at runtime, are essential to keep workers busy and avoid starvation.

Knowledge transfer further complicates matters for the following reasons:

1. New knowledge dynamically changes the shape of the search tree at runtime. This further complicates the irregular nature of the tasks, and tasks that were predicted to be long running (often those near the root of the tree) can quickly become trivial. On large systems a knowledge update can invalidate many tasks at once, requiring a quick response to redistribute work.

2. Because task runtimes, and hence number of explored nodes per task, change dynamically (due to bounding), it becomes difficult to reason about parallel performance due to the presence of performance anomalies. Performance anomalies can manifest in many ways, for example, as superlinear-speedups, and are discussed further in Section 2.3.2.1.

3. Gaining new knowledge quickly is highly beneficial to overall search time as it allows the total search space to be reduced. As heuristics guide search to promising areas, they essentially form an ordering on tasks. That is, an effective parallel search framework must maintain heuristic orderings as much as possible.

4. Knowledge is shared globally which can prove costly on large systems. Fortunately knowledge exchange is an optimisation, e.g. improved bounds, making an eventual consistency model appropriate, as opposed to requiring tasks to wait for new knowledge. In Section 6.4 we show that there are often few global knowledge updates.

2.3.2.1 Performance Anomalies

Search algorithms rely on search heuristics to attempt to find *useful* nodes as early as possible, where a useful node could be the target node in a decision problem, or a node with a strong bound for a branch and bound optimisation problem. To take advantage of these heuristics, search proceeds in a left-to-right order (at all depths of the tree).

For sequential search this left-to-right ordering means that any node has full information about the search tree, e.g. incumbents, from all nodes to-the-left of it. Because of this, every time a sequential search is run the search tree always looks the same, i.e. the same pruning opportunities present themselves every time.

Parallel searches relax this condition and instead *speculatively* search sub-trees without full information to-the-left. Speculatively exploring sub-trees is beneficial as it allows information

to also flow right-to-left, opening up additional pruning opportunities. However, it comes at the cost of potentially performing more search overall than an equivalent sequential search.

This speculation can lead to the following performance anomalies [47, 48, 49, 50]:

Detrimental Anomalies: When parallel speedup is less than one, that is, a parallel search can be slower than a sequential search. These occur when, due to not having perfect information to-the-left, a significant number of sub-trees that would have been pruned in a fully sequential search are explored. If the total amount of extra work done becomes greater than the parallelism can counteract, e.g. more than double the work for two workers, then a detrimental anomaly occurs. Detrimental anomalies can also occur during scaling, where the speedup for w workers is less than for $w - 1$ workers.

Acceleration Anomalies: When parallel speedup is greater than w, for w workers (i.e. super-linear speedups). Acceleration anomalies can occur when knowledge flows right-to-left, opening up more pruning opportunities that are unavailable to a sequential search. In practice this manifests itself as the search doing less overall work, allowing speedups greater than w.

In general we wish to avoid detrimental anomalies while allowing acceleration anomalies to occur (and try to encourage them to occur by maintaining heuristic orderings). In Chapter 7 we show a specialised search skeleton that carefully controls anomalies to give replicable performance guarantees.

The presence of anomalies makes it particularly difficult to reason about the scaling of search applications due to non-fixed workloads. We discuss these challenges further in Section 6.2.

2.4 Existing Parallel Space-Splitting Searches

Space-splitting search approaches can be classified based on how work is organised in the system. For example, Gendron and Crainic [42] split approaches into synchronous/asynchronous single/multi-pool based on the layout of workpools and then describe how work is generated/balanced in this setup. Trienekens and de Bruin [51] present a taxonomy focused around *knowledge*, which includes both tasks and bounds, and how it moves between *knowledge bases* (that reflect the architectural setup). The approach used here focuses on how/when tasks are generated and load balanced, and treats the architecture (i.e pool/knowledge base layout) as the secondary consideration. All three categorisations are similar in that each describe the same issues, e.g. workpool layout, load-balancing, synchronicity; but each has a different areas of focus.

We split the approaches into two classes based on when and how they generate work. *Static* approaches partially explore the search tree, usually breadth first, to generate a set of tasks that are then explored in a search phase. *Dynamic* approaches split the tree at runtime as new tasks are required.

To make it easier to contrast approaches we adopt consistent terminology throughout this section, based on that of Section 2.2, e.g. localities, workers, and steals, even if the original source uses a different term. We compare and contrast approaches by considering existing frameworks as examples. Given the large number of previous approaches, from many different domains, this is not a comprehensive review. Recent reviews of parallel search exist, e.g. [52], although these are often for a particular search domain.

2.4.1 Static Approaches

Static approaches usually operate in two phases: work generation and search. In the work generation phase the search tree is partially generated to create a set of tasks, i.e. sub-trees to be searched. In the search phase this set of tasks is mapped to workers who search their assigned sub-trees, independently or with knowledge exchange. Importantly, ignoring pruning, the task set is *static* on repeated runs.

Static approaches tend to be simple and can often be implemented without modifications to existing (sequential) solvers [53]. Supporting knowledge exchange between solvers may require some modification if we want knowledge exchange within a sub-tree search. An alternative is to only exchange knowledge after a search task completes, i.e. tasks start with updated knowledge but never update during the task execution.

Because there is no mechanism to dynamically generate more tasks, a major challenge is controlling task irregularity. That is, ideally the tasks created during work generation are of roughly the same size to avoid a single large task dominating the runtime.

Static approaches often differ in how they overcome this challenge by varying:

1. How they determine which sub-trees become tasks.
2. When they distribute the tasks to workers, e.g. upfront or on-demand.

A popular, and representative, example of a static approach is Embarrassingly Parallel Search (EPS) [54, 55]. It avoids imbalance by generating many more tasks than workers and allowing workers to take work as required rather than assigning tasks to workers upfront. In practice the search space is divided into $N \times w$ tasks, where w is the number of workers, and these are stored in a workpool on a master locality. Workers continuously take a task from the workpool and process it until no tasks remain or the problem has been proved satisfiable.

On-demand work distribution avoids situations where tasks cannot run because they are stuck behind a long running task as can occur if tasks are distributed to workers upfront.

Empirical analysis [54] shows $N = 30$ tasks per worker performs well for a range of constraint programming instances. More generally, low values of N risks load imbalance while high values of N increases the decomposition time, memory requirements for the workpool, and communication to/from the workpool. The work generation phase may be performed in parallel and is suggested when supporting large numbers of workers, e.g. in medium sized data centres [56].

As the work generation phase partially generates the search tree we can apply data-parallel approaches (Section 2.2.2) over the set of initial tasks. We can utilise this fact to use existing distributed data-parallel frameworks, e.g. MapReduce [24], which automate data management, task distribution, fault tolerance and networking. This is a route to performing parallel search in cloud environments which are often designed with these distributed data-parallel frameworks in mind.

For example, Xiang et. al. [57] use MapReduce [24] to solve the maximum clique problem. To achieve load balance, instead of generating a fixed number of tasks, e.g. based on number of workers, tasks are generated until their predicted runtime is deemed to be small. The runtime prediction is domain-specific, based on the number of vertices and density of sub-graphs, limiting the generality of the approach. The runtime prediction works well for a selection of random graphs (see Figure 2 of [57]), but it is unclear how well this works for practical instances. Unfortunately asynchronous knowledge transfer is not directly supported in the MapReduce model and requires custom work-arounds (i.e. using sockets) that breaks built-in fault tolerance features.

The maximum clique problem has also been solved in cloud environments by Elmarsy et. al. [58] using Apache Spark [25]. As with Xiang et al. the tasks are determined by runtime prediction. A benefit over MapReduce is that, instead of generating all partitions upfront, they allow partitioning and solving over multiple phases. Each phase consists of partitioning the graph until the (predicted small) tasks fit into memory (to avoid costly disk operations) and then storing any subgraphs requiring additional partitioning for later processing. The tasks generated in the partitioning phase are then searched and, once all tasks are complete, another partitioning phase takes place and the process repeats. This is different from the single generate-and-search model that many static approaches use, however because the runtime prediction is fixed this approach generates the same tasks on repeated runs. That is, the approach is static with on-demand task generation.

Similar task runtime predictions have been used to improve load-balancing. Imbrahim et al. [59] use statistical random sampling, while Otten and Dechter [46] solve AND/OR branch and bound problems using a machine learning approach that estimates the complexity of a

sub-problems based on previous instances. It remains unclear how well they work for a range of (domain-independent) applications.

Cloud environments have also been used to support Integer Programs using Everest [60], which generates the an initial workload via a user defined decomposition script (allowing it to adopt domain knowledge), and cOSPREY for computational protein design using MapReduce [61]. cOSPREY uses a pipeline of MapReduce jobs where each job generates a new set of inputs for the next by decomposing the problem further.

Cube-and-Conquer [62] is a static approach to solving boolean satisfiability problems (SAT). Here, cubes (partial variable assignments) are created upfront using look-ahead SAT solvers [63] before being solved in parallel using incremental SAT solvers (CDCL). The amount of work generated is based on predicting task runtime as the product of the number of decisions and number of assigned variables in the cube; a domain-specific heuristic.

Fischetti and Monaci [64] present SelfSplit, an approach that does not require a centralised workpool and hence minimises communication between workers. In SelfSplit *every* worker decomposes the search tree into a fixed task set and a deterministic algorithm maps these tasks to a specific worker. Workers ignore the other tasks that are not assigned. Assigning tasks to workers uses a domain-specific (constraint programming) task complexity estimate and round-robin scheduling to attempt an even distribution of tasks to workers. Without a centralised workpool there is no method to rebalance if the task distribution proves to be poor.

2.4.1.1 Static Approach Summary

Static approaches are an efficient form of parallelism. They work well in distributed environments as the amount of communication is small, e.g. a single node on-demand. This allows them to often scale to large numbers of workers, e.g. 800 cores for (two level) Cube-and-Conquer [65], 512 workers for EPS [56] and 100 workers for MapReduce Maximum Clique [57].

Many approaches rely on domain-specific heuristics to determine when to stop generating tasks that cannot be implemented in a general-purpose parallelism framework without a method for a user to provide this information. In Section 4.3.4 we introduce a general-purpose static approach that spawns tasks until a fixed cut-off depth in the search tree. We avoid scalability issues by opting for distributed workpools rather than a centralised master workpool. Although simple, these approaches can lead to good performance as illustrated in Section 6.5.

As the task set for a given instance is fixed many static approaches allow for deterministic results, i.e. they always find the same result even if there are multiple solutions. If they are order preserving and perform bounds sharing they can be used to avoid performance anomalies

(Section 2.3.2.1). We explore this idea further in Chapter 7 and show how a skeleton based on a static approach allows replicable performance guarantees.

2.4.2 Dynamic Approaches

Dynamic approaches divide the search space during search rather than in a predefined work generation phase. These approaches can be classified by their load-balancing strategy: periodic, centralised, or decentralised.

2.4.2.1 Periodic Load-Balancing

Periodic load-balancing approaches interspace search phases that process existing tasks, with load-balancing phases that generate new tasks if necessary. Periodic approaches differ in:

1. The frequency of load-balancing phases.
2. Determining which workers receive work.
3. Determining how much work should be given to another worker.

Karp and Zhang [66] (and more recently Pietracaprina et al. [67]) divide search into three phases: (depth-first) *traversal*; *pairing*, where idle processors are matched with busy workers; and *donation* where work is moved between paired workers. Workers are paired either deterministically, via a maximal matching algorithm, or randomly. In [67] donation sends the "topmost unexplored right subtree" (assuming binary search trees) to the paired worker, while in [66] half the nodes at the lowest depth are donated. The algorithms are shown to be theoretically efficient but not empirically analysed.

Sanders [68] uses a poll-and-shuffle approach to load-balancing. Originally designed for hypercube architectures [69], search is divided into *cycles* where each cycle consists of multiple work phases. At the end of work phase i, work requests (if required) are sent down dimension i of the hypercube. At the end of a full cycle, subproblems are randomly permuted across all workers in order to avoid clusters of work forming. While designed for hypercubes the approach can be generalised to other networks by selectively choosing neighbours at the end of each work phase. The approach is only analysed theoretically, making it unclear how this approach performs in practice.

Periodic load-balancing has also proved useful for single-instruction-multiple-data (SIMD) machines where asynchronous load-balancing is not possible. For example, Karypis and Kumar [70] use a dynamic triggering scheme to determine when to initiate load-balancing. In this scheme load-balancing is triggered when the idle time of processors during the search

phase and (predicted) cost of the next load-balancing phase are equal. Sharing of work is performed by matching idle workers to busy workers, in a round-robin fashion, starting from the processor *after* the last to donate work (to avoid the same processors always donating). The work to be donated is created dynamically by partitioning unexplored nodes into two parts. While SIMD computers have gone out of fashion[12] the triggering and load-balancing ideas could be utilised by other periodic re-balancing systems, and perhaps adopted by GPU based search frameworks.

DryadOpt [71] runs all workers run until a predefined time deadline. Like many static approaches, DryadOpt targets cloud environments by making use of Dryad [72], a data-parallel compute engine similar to MapReduce or Apache Spark. DryadOpt is designed with generality in mind and allows domain-independent branch and bound searches to be implemented.

The tree search framework `mts` [73] form of *asynchronous* periodic load-balancing. In this scheme workers are given a sub-tree and *budget* (number of traversed nodes) by a master. Workers then search the sub-tree until either the budget is exhausted or the search is complete and then return any unprocessed nodes to the master. The master collects unprocessed nodes and assigns these on-demand to idle workers. In this way the budget, and sub-problem, forms a period; though not all workers will complete their searches at the same time.

In Section 4.3.6 we introduce an approach similar to `mts` that re-partitions work based on a *backtracking budget*. Our approach features fully distributed workpools and work-stealing rather than a centralised master process.

2.4.2.2 Centralised

The most common approaches to space-splitting search use asynchronous load-balancing where work is requested/assigned as it is required. A common software architecture is master–worker, where the master operates as a centralised load manager and solution store. Master–hub–worker architectures are extensions of master–worker where a master manages several hubs that in turn manage several workers. Centralised schemes are particularly useful for performing best-first search as they can maintain a global task ordering.

The BOB family of frameworks (BOB [74], Bobpp/Bob++ [75], and Ibobpp [76]) centre parallelism around the notion of a *global priority queue (GPQ)*. Each search step consists of removing an unprocessed node from the GPQ, processing it, and adding any remaining children back into the GPQ. This approach is known as node-oriented search [76]. The load-balancing between processes is handled transparently by the GPQ, which, by being

[12] SIMD computing with GPUs is currently popular, however it is not clear that the architecture lends itself well to search due to the large number of branching instructions often encountered.

an abstract type, allows interchanging of methods. For example, Bobpp supports the Kaapi task-parallel environment [77] where inserts into the GPQ correspond to spawning a new task[13]. Likewise concurrent priority queues could be used to implement efficient shared-memory parallel approaches. One difficulty with node-oriented search is that the GPQ can become a bottleneck, particularly with a large number of workers and/or short node processing functions. To overcome this Ibobpp [76] suggest using node-oriented search to generate many tasks and then reverting to a sequential[14] search. While similar to a static approach, by introducing a dynamic threshold that is based on the number of waiting workers, work is dynamically re-partitioned at runtime. Similar node-orientated approaches have been used for both distributed-memory [78] and shared-memory applications [79, 80], including in the Muesli skeleton framework [81] .

MaLLBa [82], a skeleton library for combinatorial optimisation problems (both exact and approximate), makes use of a master–worker [83] scheme where the master tracks the state of workers (e.g. idle). Workers that require help to complete a large sub-tree can ask the master for a list of idle workers to push work to. The pushing of work is worker–to–worker and does not go through the master. Schulte [84] uses a similar approach however this time the *idle* workers ask the master (called a *manager*) for work. The master then selects a busy worker and requests it shares some work with the idle worker.

The approach of Dabah et al. [85] uses a multiple workpool approach. The master maintains a global workpool that is populated via a breadth-first traversal of the tree (to some depth). Each worker likewise maintains a current workpool of locally generated nodes (traversed depth-first). If a workers local workpool is empty it requests work from the global workpool. In the case that the global workpool is also empty the master gathers one node from each busy worker; refreshing the global workpool. While not explored in [85], the use of local workpools would allow each worker to itself run in parallel (on multi-core say) essentially creating a master–hub–worker configuration.

Master–Hub–Worker The scalability of master–worker schemes is limited by the performance of the master process, i.e. it is a potential bottleneck. To overcome this additional hub processes can be introduced that manage a subset of available workers in a hierarchical fashion.

The PICO framework [86] and its predecessor PEBBL [87] make use of such an architecture to solve integer programming problems. As in many master–worker schemes the hubs maintain a workpool of unexplored nodes and use this to push work to idle workers (in this case by requesting the worker who owns the open node perform the donation). An interesting feature

[13] As Kaapi supports distributed workpools, Bob++ may also be classified as a decentralised. We classify it as centralised here as the core model is based around a *global* queue.

[14] Allowing communication of new solutions/bounds.

of PICO is that nodes (more precisely node identifiers) are probabilistically released to the hub, e.g. a probability of 100% implies all new nodes are given to the hub. These probabilities are set based on system parameters such that workers with more work donate more often. If the hubs run out of work then they may request a task from each worker in a similar manner to Dabah et al. [85]. Two mechanisms support load-balancing between hubs, *scattering* where workers release nodes to a remote hub instead of the local hub, and *rendezvous* where the root master receives workload information from all hubs and uses this information to pair hubs based on workload such that workload is be equalised across the system (tree based load-balancing). Rendezvous is a form of periodic load-balancing showing how hybrids of the dynamic approaches to parallel search can be used to good effect. ALPS [88] is a more recent attempt to apply master–hub–worker schemes to integer programming that adopts many ideas from PICO/PEBBL.

Tree-based load-balancing procedures are likewise used by Jaffar et al. [89], that asks all workers to predict how much work they can provide and uses this to match idle to busy workers. Vu and Derbel [90] also use tree based load balancing based on relative sub-tree size, i.e. difference in number of workers per hub. To avoid clustering of work a scattering approach is used allowing localities to send work requests through random "bridges" to other localities.

An additional benefit of centralised approaches, not explored in this work, is the possibility of providing fault tolerance[15], e.g. [91], by tracking who requested specific sub-problems and restarting these in case of node failure.

2.4.2.3 Decentralised

Workers may also operate fully asynchronously, maintaining load balance in a distributed fashion, often using work-stealing. Important design decision in the decentralised approach include victim selection, i.e. which worker should be asked for work, and determining a suitable workload to send on a request.

These approaches are often based around dynamically splitting a search tree when a work request is received. This can be seen as an application-level work-stealing scheme that, instead of stealing existing tasks from workpools, also provides the *creation* of tasks.

One of the first parallel search systems to feature a fully distributed dynamic approach was DIB [92] (A Distributed Implementation of Backtracking). In this approach each worker maintains two tables, *WorkGotten*, that tracks nodes remaining to be processed, and *WorkGiven*, that is used to determine when a stolen node has been processed by another worker. In modern task-parallel environments, WorkGotten is equivalent to a workpool while WorkGiven is often

[15] Fault tolerance can be provided in a distributed scheme, but doing so is typically more complex.

represented using (distributed) future objects. Two victim selection policies are available. The first organises the workers into a ring and propagates steal messages around the ring until work is found. Propagation starts from the successor of the last successful steal (similar to the approach of Karypis and Kumar [70] in their periodic scheme) to avoid always stealing form the same workers. The second policy broadcasts requests to k machines at a time without request forwarding. Upon receiving a work request a worker will either send a portion from the local workpool (WorkGotten), or will split the tree it is current exploring (if it is known to be non-trivial). Muesli [81] supports a similar ring based stealing scheme where work requests are sent to a direct neighbours only. Victims send their second best problem (i.e. the one they are not working on) on receipt of a work request as this is most likely to generate more work.

Sanders [93] presents a similar scheme where trees are always split (if possible) on a steal request. Victim selection is done at random to avoid problems with ring based work-stealing schemes, i.e.

> "The basic problem of these neighbourhood polling schemes is that highly loaded PEs are quickly surrounded by a cluster of busy PEs and are therefore unable to transmit work" [93].

The Cilk [34] developers likewise analyse the advantages of a randomised victim selection scheme.

A random work-stealing approach has also been applied to maximal clique enumeration [8], and tested using different work splitting mechanisms: steal high, steal low, or steal vertical split [94]. Zhang et al. [95] provide a theoretical analysis of randomised work-stealing for tree search where a single top-most node is given to the thief.

Vu and Derbel [96] likewise use random work-stealing to enable load balance, but, to support heterogeneous machines, the amount of work transferred is based on the relative power of the workers, where power is defined as the number of nodes processed per second. By using nodes per second instead of a measure such as clock speed they allow for architectures featuring low clock speedups but high throughput, e.g. GPU acceleration, where the combined performance might be higher than a single faster CPU.

ZRAM [97] mixes random work-stealing with ring based steals such that idle workers first select victims at random. If a steal is unsuccessful, then victims are chosen in a round-robin fashion (forwarded from the last victim node). Justification is given that this keeps the "algorithm simple", presumably meaning that a list of already chosen victims does not need to be forwarded with the steal message.

In Section 4.3.5 we detail an approach that relies on random distributed work-stealing and on-demand tree splitting to perform parallel search.

Locality Load Information To further improve load-balancing some systems maintain information about the load of neighbouring localities.

Lüling et al. [98] use a weight function (based on the bounds of unexplored nodes) to determine the load of a worker. Load-balancing is then triggered based on detecting a changing load. For example, large increases of load cause work-pushing to neighbours, while large decreases in load cause work requests to be sent to neighbours.

Gau and Stadtherr [99] present an advanced decentralised load-balancing approach where state information is periodically shared between workers. This state information uses stack length as a representation of work-load and, using this, work-stealing (or work-pushing) can be performed in a principled manner, i.e. with more knowledge than random work-stealing. Such an approach performs well in practice for a small number of workers (16). Di Fatta and Berthold [100] adopt a similar approach where workers record workload information in both a peer-to-peer and centralised fashion. Instead of stack length, they use starting time of the current job as a heuristic and steal the longest running jobs first.

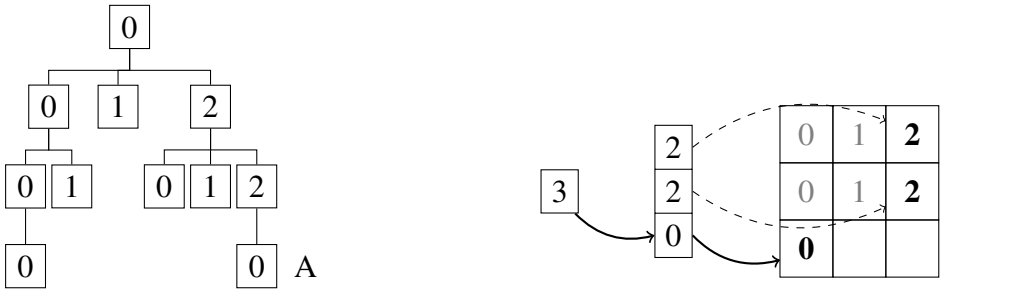
Confidence based work-stealing [101] is a scheduling algorithm specifically designed for parallel constraint programming. The algorithm is based on estimating solution density of sub-trees and using this to assign more workers to fruitful areas of search. Confidence estimation is performed online allowing the approach to adapt to any problem instance. The work-stealing algorithm does not actually *steal* work. Instead, workers restart from the root node and use the confidence values to traverse the current search tree until an unexplored node (with high confidence) is encountered, i.e. it "steals" unexplored sections of a global tree. This requirement of having a global search tree limits the approach to shared-memory, although it would be possible for stolen sub-trees to be sent and solved in parallel on other localities. In this scheme workers do not need to fully explore their assigned sub-trees. Instead a budget measure (i.e. a periodic approach) is used to tell the workers when to restart work-stealing. This avoids workers getting stuck in areas that prove to have limited confidence.

2.4.2.4 Index-Based Stealing

Most approaches are based around workpools of unexplored nodes or a splitting function that can dynamically produce set of nodes. However, it is also possible to use indices into the search tree as a unit of work.

Given a search tree, and a deterministic branching function, it is possible to uniquely identify each node in the tree using the path from the root to the node. For example, node A in Figure 2.4(a), is assigned label 0220.

Abu-khzam et al. [102] make use of this technique to implement a work-stealing scheme based on stealing indices. During search each worker keeps track of the current position in



(a) Nodes as Tree Indices (b) Integer-Vector-Matrix Approach.

Figure 2.4: Indexed approaches.

the tree and how many child nodes are available at all depths above. When a worker receives a steal request the largest task can be calculated quickly by finding the lowest depth that has child nodes remaining. The index of the lowest depth child node is sent to the thief without ever needing to generate the node locally. The thief then recomputes the tree using this path to find the starting position. This approach supports stealing multiple nodes/paths if there are multiple at the lowest depth[16].

We use a similar approach in Section 4.3.5 of exploring the stack to find the most promising next node. However, we sends the node itself rather than search indices to avoid recomputes (although could be extended to support both).

Mezmaz et al. [103] show how combinatorial problems based on permutations can be described using an $N \times N$ matrix and a position vector, where N is the number of objects, e.g. graph vertices. This is known as the Integer-Vector-Matrix (IVM) approach and is shown graphically in Figure 2.4(b). An IVM consists of a depth integer, a vector that details the path through the tree (e.g. 220 above), and a matrix that represents branching choices. With the IVM searches can be described by position intervals, rather than nodes, e.g. worker 1 can explore from 0000 to 2000 while worker 2 explores 2100 to 3210. When work requests occur the interval can be dynamically split, although a specific interval might contain no work. A key advantage of this approach is the reduced and deterministic memory usage as the structures are fixed size. This approach has been applied in the context of GPU search [43, 44], where managing linked list structures is difficult, but vector and matrix updates are much easier. So far IVM has only been used for Flowshop instances and it is unclear how well it applies to non-permutation problems.

2.4.2.5 Dynamic Approaches Summary

The main benefit of dynamic approaches is their ability to create new work at runtime in respond to system conditions, e.g. starvation. Implementations are typically more complex to

[16]In the original implementation stealing N nodes must from the right-most child. While simple to implement, this goes against the heuristic ordering which suggest the left-most unexplored node is a better choice.

manage the increased communication both work and control messages and to introduce work splitting methods. By utilising techniques such as random work-stealing deterministic results are no longer guaranteed.

There are three main types of dynamic approaches: 1) periodic load-balancing, which iterates between search and load balance phases; 2) centralised schemes, which introduce master localities that accumulate knowledge to make load-balancing decisions; and 3) fully decentralised approaches which often take the form of work-stealing scheduling to handle the irregularlity.

2.4.3 Scalability of Existing Approaches

There have been many attempts to scale parallel combinatorial search to large architectures over the years. Table 2.1 highlights the scale achieved, and the application domains considered, by the approaches outlined in Section 2.4.1 and Section 2.4.2. We report the scales and domains found in the literature. This does not imply that approaches could not scale further, or be used in additional application domains.

Given the wide range of architectures and, in turn, wide range of worker performance, it is difficult to compare approaches based solely on number of workers. It does however show that scalable approaches are possible in practice even given the challenges of parallel search (Section 2.3.2).

No one style of parallelism dominates high worker counts, although master–hub–worker approaches do appear to use higher numbers of workers than master–worker approaches showing that the single master bottleneck can be effectively removed. Abu-Khzam et al. [102] utilise the largest amount of workers (131,072) overall and show that distributed work-stealing based approaches can scale.

Likewise, no one application/domain dominates, and many frameworks are only tested on a single application/domain. YewPar (5), described in this work, supports the largest number of applications/domains (6)[17], with DIB [92] supporting the next largest number of applications/domains (5).

2.4.4 The Need For A New Search Skeleton Framework

This section motivates why, given the range of existing search approaches and parallelism frameworks, a new skeleton based search framework, YewPar (Chapter 5), is required.

Task-parallel frameworks provide many features that aid parallel tree search, in particular work-stealing to handle the irregularity of search. However many make assumptions that are invalid

[17] 7 if we consider the decision clique search variant to differ from optimisation.

Framework	Approach	System	Workers Reported	Application/Domain Reported
Embarrassingly Parallel Search [56]	Static	Data Centre	512	Constraint Programming
Xiang et al. [57]	Static	Data Centre	128	Maximum Clique
Elmasry et al. [58]	Static	Data Centre	32	Maximum Clique
Everest [60]	Static	Cluster	26	Mixed Integer Programming
Cube-and-Conquer [62]	Static	Supercomputer	800	SAT
SelfSplit [64]	Static	Shared-Memory	64	Constraint Programming
Karypis and Kumar [70]	Periodic	Connection Machine 2	32,000	n-Puzzle
DryadOpt [71]	Periodic	Cluster	512	Shared/Distributed-memory
mts [73]	Periodic	Cluster	192	Steiner Trees
BOB [74]	Centralised	Cluster	5	Quadratic Assignment; Vertex cover
Bob++ [75]	Centralised	Cluster	184	Quadratic Assignment; Mixed Integer Programming
Ibobpp [76]	Centralised	Cluster	24	Quadratic Assignment; n-Queens
Schulte [84]	Centralised	Cluster	6	Constraint Programming
Debah et al. [85]	Centralised	Cluster	350	Job-shop Scheduling
PEBBL [87]	Master–Hub–Worker	Supercomputer	8,192	Mixed Integer Programming
ALPS [88]	Master–Hub–Worker	Cluster	32	Mixed Integer Programming
Jaffar et al. [89]	Master–Hub–Worker	Cluster	62	Integer Programming
Vu et al. [90]	Master–Hub–Worker	GRID'5000	1,000	Flow-shop Scheduling; Unbalanced Tree Search
Bendjoudi et al. [45]	Master–Hub–Worker	GRID'5000	8,900	Flow-shop Scheduling
DIB [92]	Distributed	Multicomputer	20	8-Queens; Knights Tour; Travelling Salesperson; Petri Nets Reachability; Knapsack
Muesli [81]	Centralised/Distributed	Cluster	16	n-Puzzle; Travelling Salesperson
Schmidt et al.[8]	Distributed	Supercomputer	2,048	Maximal Clique Enumeration
Vu et al. [96]	Distributed	GRID'5000	512	Flow-shop Scheduling
ZRAM [97]	Distributed	SuperComputer	150	Convex Hull; 15-Puzzle
Lüling et al. [98]	Distributed	Unknown	1,024	Vertex Cover; Travelling Salesperson; n-Puzzle; VLSI Layout
Gau et al. [99]	Distributed	Cluster	16	Vapor-Liquid Equilibrium
Difatta et al. [100]	Distributed	Cluster	24	Subgraph Mining
Schulte et al. [101]	Distributed	Shared-Memory	8	Constraint Programming
Abu-Khzam et al. [102]	Indexed	Supercomputer	131,072	Vertex Cover; Dominating Set
Mezmaz et al. [103]	Indexed	Shared-Memory	16	Flow-shop Scheduling
YewPar	**Static/Periodic/Distributed**	**Cluster**	**255**	**Unbalanced Tree Search; Numerical Semigroups; Subgraph Isomorphism; Clique Search; Knapsack; Travelling Salesperson**

Table 2.1: Summary of existing approaches: scalability and application domains.

for search problems. For example, standard deque based work-stealing can break heuristic search orderings (discussed in Section 4.4). Likewise, many work-stealing frameworks assume that the number of tasks in a workpool is a good measure of worker/locality load. As pruning can make many tasks trivial this is not necessarily the case in search.

Many work-stealing algorithms also assume that tasks are already created and should be balanced between nodes. For tree search applications we often want to dynamically split the work (Section 2.4.2) which requires some interaction at application level. Ideally this, and knowledge exchange, should be hidden from the user, requiring search specific frameworks.

Many existing search approaches are designed with a particular application in mind, e.g. Integer programming in PICO [86], PEBBL [87] and ALPS [88], or constraint programming with EPS [54] and confidence based work-stealing [101]. This can manifest itself as, for example, domain-specific functions for predicting task sizes, or many frameworks assuming optimisation problems without providing support for enumeration or decision variants.

Likewise, many approaches are designed with a particular scale in mind, often distributed or shared-memory but seldom both. This is likely an historical artefact as many approaches were designed before the widespread adoption of multi-core processors. Older approaches are also based on a set of assumptions, such as inter process communication being particularly costly, that are less true today[18].

While the search approaches, by implementing a backtracking search algorithm, could support any search application, there is often no general-purpose API for search. This causes the frameworks to be difficult to extend and makes it difficult to compare approaches. Few papers describing an approach will make performance comparisons with others, a notable exception is [79]. Finally, most approach implementations are not openly available making them difficult to adopt for new search problems.

Two frameworks that attempt to provide a high-level unified approach to parallelism using skeletons are Muesli [81] and MaLLBa [82]. Both frameworks are designed for branch and bound optimisation problems, and unlike YewPar, do not currently support decision or enumeration searches. Being skeleton based, different parallelism approaches can be used with the same domain-specific search application. However, both frameworks provide a limited selection of parallel approaches. MaLLBa implements a single master–worker approach, while Muesli supports both a centralised scheme and a distributed scheme based on work-stealing in a ring. The virtual class hierarchy approach of Muesli has been shown to have high overhead [104]. Bob/Bobpp/Ibobpp [74, 75, 76] similarly provides a separation between

[18]While inter process communication is still costly today relative to shared-memory access, it is typically much faster than a search task.

user search and implementation details by utilising the global priority queue abstraction.

A new framework can overcome these issues. It can extend standard task-parallel functionality, e.g. spawn and futures, to support search specific work-stealing and task generation, as well as knowledge exchange. By adopting a consistent API that covers all three types of search problem, we allow many different parallelism approaches to be both implemented and compared. YewPar (Chapter 5) is open source [105], supports three types of search (enumeration, decision and optimisation), and features a wide range of parallel approaches inspired by the literature. In particular: Depth-Bounded (Section 4.3.4), a static approach that converts any node below a user specific d_{cutoff} to task; Stack-Stealing (Section 4.3.5), a dynamic approach based on random work-stealing between workers; and Budget (Section 4.3.6), an asynchronous periodic approach that spawns new tasks once a user specified number of backtracks has been performed.

Chapter 3

Formalising Parallel Tree Search

This chapter formalises parallel, space-splitting, backtracking tree search as small-step structural operational semantics [106]. The family of models, referred to as MT^3 (*Multi-Threaded*[1] *Tree Traversal)*, is defined in an abstract, domain-independent manner, allowing reasoning about high-level (i.e. non domain-specific) parallel tree search.

MT^3 follows the BBM model of Archibald et al. [4] that models trees as partially ordered sets (Section 3.1) and shows how, via an order-isomorphism from words over $\mathbb{N}$, this partial order can be extended to a total ordering allowing search order heuristics (Section 2.1.4.1) to be encoded (Section 3.1.1). In particular we add (direct) support for enumeration and decision search (Section 3.3.6), spawn rules for generating new tasks (Section 3.3.8), and an ordering on reductions ensuring correctness (Section 3.3.2).

Modelling parallel search as operational semantics is novel. Previous approaches are designed to prove properties of performance anomalies [47, 48, 49, 50] rather than specifying general parallel search. These models do not give full reductions over search trees and generally model search as a set of tasks where, at each step, (a subset of) tasks are removed, processed, and children reinserted (i.e. node-oriented search). These assumptions do not allow the range of existing approaches Section 2.4 to be modelled, e.g. they do not allow explicit spawning of tasks.

MT^3 maintains a separation of concerns that make it easier to vary both the type of search and style of parallelism. A MT^3 model consists of four rule categories:

1. **Traversal** rules (Section 3.3.4) that specify a) how tasks are assigned to threads, b) how to move through a tree in a depth-first manner, and c) the conditions for terminating a thread.

[1]The use of multi-threaded here implies the ability to have multiple, individually scheduled workers, rather than requiring a physical implementation to use a threading abstraction. As such, the model may be applied to any worker configuration, including distributed-memory.

2. **Node Processing** rules (Section 3.3.6) that extend the model to support a specific type of search, allowing MT^3 to support all of: enumeration, decision and optimisation searches.

3. **Pruning** rules (Section 3.3.7) that add support for branch and bound search.

4. **Spawn** rules (Section 3.3.8) that capture work generation, allowing succinct descriptions of the search skeletons described in Chapter 4.

Combinations of these rules are selected to model a particular search, e.g. an enumeration search with branch and bound pruning, but no spawning. That is, MT^3 is not a single model, but a family of closely related models.

A key component of algorithmic skeletons, such as those in Chapter 4, is the ability to specify a user computation that is independent of the parallel coordination. In Section 3.4 we show how the tree definitions used by MT^3 give rise to a suitably abstract interface for tree search allowing domain-specific searches to be passed to the skeletons.

3.1 Modelling Search Trees

MT^3 is derived from the BBM model of Archibald et al. [4] who present a formalisation of parallel tree search based on trees as partially ordered sets and similarly provide small-step operational semantic reduction rules that specify parallel state transitions. Unfortunately, BBM does not capture all the tree searches presented in this work, as BBM:

1. Does not capture how work is spawned; instead it is always constructed *a priori*.

2. Only provides direct support for optimisation searches with no support for enumeration problems. Decision searches are possible in BBM by encoding early termination in the pruning function, but not made explicit in the model itself.

3. Assumes branch and bound searches and does not directly support searches that do not perform pruning without introducing a trivial pruning predicate.

This work generalises and extends BBM to support additional search types (Section 3.3.6), non branch and bound searches, and task spawns (Section 3.3.8).

Adopting the terminology of Archibald et al. [4] search trees may be formalised as follows.

Let X be a non-empty set with power set 2^X. The set of finite words over alphabet X is denoted by X^*, with the empty word denoted by ϵ. $|w|$ denotes the length of a word $w \in X^*$, where $|\epsilon| = 0$.

We denote the prefix order on X^* by $\preceq$ and use $\leq_{\text{lex}}$ to denote the lexicographic extension of the natural order $\leq$ on $\mathbb{N}$ to $\mathbb{N}^*$. A prefix order is a reflexive and transitive substring relationship on words such that $u \preceq v$ if u is a prefix of v, that is, there is some w such that $uw = v$. For example the words cc and $ccde$ both both have cc as a prefix and hence $cc \preceq ccde$. $\leq_{\text{lex}}$ is an extension of the prefix order $\preceq$, such that, for two words, e.g $m = 001$ and $n = 0023$, $m \leq_{\text{lex}} n$ compares the words numerically based on the first position that does not match a common initial substring. That is, $m \leq_{\text{lex}} n$ as the first non-common prefix comparison gives $1 \leq 2$. Lexicographical ordering gives a total ordering on the set $\mathbb{N}^*$, e.g. $0 <_{\text{lex}} 00 <_{\text{lex}} 1 <_{\text{lex}} 10$. We define a function $succ(S, w)$ that returns the (unique) element that comes strictly after w in $\leq_{\text{lex}}$ order in a finite non-empty set $S \subset \mathbb{N}^*$ if it exists, else $succ(S, w)$ returns $\perp$.

Trees are defined as prefix-closed sets of words. A tree T over alphabet X is a non-empty subset of X^* such that there is a least (w. r. t. $\preceq$) element $u \in T$ and T is prefixed-closed above u. A tree is prefixed-closed above u if for all $v, w \in X^*$, $u \preceq v \preceq w$ and $w \in T$ implies $v \in T$. We use the notion $T = (X, \preceq)$ to denote a tree over X with ordering $\preceq$.

Elements of T (i.e. $v \in X^*$) are called *nodes*[2] with the least element $u \in T$ known as the *root*. A maximal node v, where there is no u in T such that $v \preceq u$, is known as a *leaf*. Given nodes u and v, if $|v| = |u| + 1$ and u is a prefix of v then we call u a direct *child* of v. The set of all direct children of u is given as $children(u) = \{v \in T \mid |v| = |u| + 1 \wedge u \preceq v\}$.

Given a node $u \in T$, a sub-tree S rooted at u is the set of all vertices sharing the prefix u in T. That is, $S \subseteq T$ where u is the minimal element of S and the prefix order, $\preceq$, of elements in S is maintained (i.e. if $u \prec v \prec w$ in T then $u \prec v \prec w$ in sub-tree S). The sub-tree rooted at a node u is given by $subtree(T, u) = \{v \in T \mid u \text{ is a prefix of } v\}$. u is a prefix of itself and, as such, also appears in $subtree(T, u)$.

An example tree over the natural numbers is given in Figure 3.1. Each node is labelled based on the path from the root to that node. These paths are known as *branches* in the tree.

Here the node set = $\{\epsilon, 0, 1, 2, 00, 01, 20, 21, 000, 210, 211\}$, with $\epsilon \leq_{\text{lex}} 0 \leq_{\text{lex}} 00 \leq_{\text{lex}} \cdots \leq_{\text{lex}} 211$. An example sub-tree S is $\{0, 00, 01, 000\}$ where 0 is the root of the sub-tree and 000 and 01 are leaf nodes.

A *tree generator* is a function $g : X^* \to 2^X$. We define t_g as the smallest subset of X^* that contains ϵ and is closed under g, where closure implies that for all $u \in t_g$ and all $a \in g(u), ua \in t_g$. That is, t_g is the tree *generated* by g. For example, the *tree generator* for

[2] In Archibald et al. [4] these are known as *vertices*.

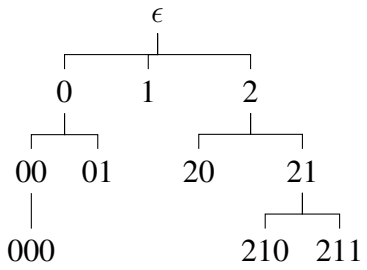


Figure 3.1: Example search tree over $\mathbb{N}$ where nodes correspond to (finite) elements of $\mathbb{N}^*$.

Figure 3.1 may be defined as the function that maps:

$$
\begin{aligned}
g(\epsilon) &= \{0, 1, 2\} \\
g(0) &= \{0, 1\} & g(00) &= \{0\} \\
g(2) &= \{0, 1\} & g(21) &= \{0, 1\} \\
g(a_0 \dots a_i) &= \{\}
\end{aligned}
$$

From the closure property we have, for example, $g(0) = \{0, 1\}$ implies t_g contains the nodes 00 and 01.

3.1.1 Ordered Trees

An important feature of decision and optimisation searches is the use of search heuristics to guide search towards solutions (Section 2.1.4.1). In a depth-first search heuristics take the form of orderings of child node exploration, i.e. good candidates should be placed further to-the-left. To capture this, we generalise trees to *ordered* trees by labelling trees as elements of $\mathbb{N}^*$ and using $\leq_{\text{lex}}$ to provide a total search order.

Formally, an *ordered tree* λ *over* X is a function $\lambda : (\mathbb{N}, \preceq) \to (X, \preceq)$ such that λ is an order isomorphism between the two trees. This implies, $\forall u, v \in (\mathbb{N}, \preceq)$ if $u \preceq v$ then $\lambda(u) \preceq \lambda(v)$.

We overload the λ notation to denote both this ordered tree function above and the corresponding image of the function (i.e. the tree over X). That is, we call λ an *ordered tree* over X.

An example ordered tree is given in Figure 3.2. Each node in $(\mathbb{N}, \preceq)$ has a corresponding node under the image of λ, for example $00 \to aa$ and $1 \to c$.

As λ is an order isomorphism, the lexicographical ordering, $\leq_{\text{lex}}$ on $\mathbb{N}^*$, carries over to the ordered tree λ. This gives a *total ordering* on λ such that $0 \leq_{\text{lex}} 00 \leq_{\text{lex}} 01 \leq_{\text{lex}} 1 \leq_{\text{lex}} \cdots \leq_{\text{lex}} 21$ implies $a \leq_{\text{lex}} aa \leq_{\text{lex}} ab \leq_{\text{lex}} c \leq_{\text{lex}} \cdots \leq_{\text{lex}} bb$. To avoid confusion we call the elements of $(\mathbb{N}, \leq_{\text{lex}})$ *positions*, although formally these are tree *nodes*.

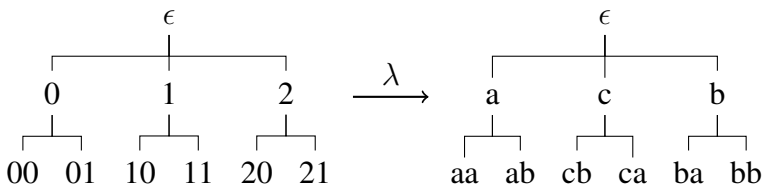


Figure 3.2: Example ordered tree. Each position in the left tree corresponds to a node on the right (the image under λ). For example $00 \to aa$ and $1 \to c$.

An *ordered tree generator* is a function $g : X^* \to X^*$ such that all images of g have no repeating letters (are isograms). We can define an ordered tree generator $\lambda_g : (\mathbb{N}, \leq_{\text{lex}}) \to X^*$ as the function with the smallest domain (i.e. the smallest subset of $\mathbb{N}^*$) such that:

- $\lambda_g(\epsilon) = \epsilon$
- λ_g is closed under g.

Closure under g implies that for all positions $u \in (\mathbb{N}, \leq_{\text{lex}})$ and nodes $v = \lambda_g(u)$, if $g(v) = a_0 a_1 \dots a_{n-1}$ and $i < n$ then ui is a position in $(\mathbb{N}, \leq_{\text{lex}})$ and $\lambda_g(ui) = va_i$.

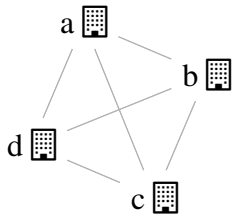
By construction λ_g is an order isomorphism and hence an ordered tree, the ordered tree *generated* by g. For example, the *tree generator* for Figure 3.2 is the function that maps:

$$\begin{aligned} g(\epsilon) &= acb & & \\ g(a) &= ab & g(c) &= ba \\ g(b) &= ab & g(d_0 \dots d_i) &= \epsilon \end{aligned}$$

Where $g(c) = ba$ implies that nodes cb and ca are present in the tree and that $cb \leq_{\text{lex}} ca$ by extension of $\leq_{\text{lex}}$ under λ (i.e $10 \leq_{\text{lex}} 11$ in $(\mathbb{N}, \leq_{\text{lex}})$).

3.2 Example: Tree Generators for the Travelling Salesperson Problem

To show how the abstract terms defined above map to a specific search tree we use the famous travelling salesperson problem as an example. The travelling salesperson problem (TSP) searches for the shortest tour between N cities that returns to the (fixed) starting city. An implementation of TSP is discussed in Section 5.2.3.2. This formulation of TSP comes largely from Archibald et al. [4].



	a	**b**	**c**	**d**
a	0	3	2	5
b	3	0	1	6
c	2	1	0	2
d	5	6	2	0

Figure 3.3: Example (symmetric) TSP instance.

The input to TSP consists of a set $\mathcal{C}$ of n cities and a cost function given by the symmetric non-negative distance function $d : \mathcal{C} \times \mathcal{C} \to \mathbb{R}$. Figure 3.3 shows an example with four cities and associated cost function given by the distance matrix (e.g. $d(a, b) = 3$).

Tours are modelled as isograms over $\mathcal{C}$ where the word $t = c_1 c_2 \dots c_k \in \mathcal{C}^*$ represents a (partial) *tour* starting at c_1 and ending at c_k. The tour is complete if $k = n$, that is, if every city in $\mathcal{C}$ is visited exactly once. Complete tours implicitly include the return to the root city in their distance calculations.

The distance function d can be generalised to operate on words, $\mathcal{C}^*$, such that:

$$\begin{aligned} d(\epsilon) &= 0 \\ d(c_1) &= 0 \\ d(c_1 \dots c_k c_{k+1}) &= d(c_1 \dots c_k) + d(c_k, c_{k+1}) \\ d(c_1 \dots c_n) &= d(c_1 \dots c_{n-1}) + d(c_{n-1}, c_n) + d(c_n, c_1) \end{aligned}$$

An unordered tree generator, $g : \mathcal{C}^* \to 2^{\mathcal{C}}$, extends a partial tour with every city that has not yet been visited, enumerating all possible tours.

$$g(c_1 \dots c_k) = \mathcal{C} \setminus \{c_1, \dots, c_k\}$$

For example, in the TSP instance of Figure 3.3, given the (partial) tour ac, $g(ac) = \{b, d\}$ implying that acb and acd are both nodes in the tree generated by g.

The tree is built be repeatedly applying g, fixing a starting city[3] of a.

$$\begin{aligned} g(\epsilon) &= \{a\} & g(a) &= \{b, c, d\} & & \\ g(ab) &= \{c, d\} & g(ac) &= \{b, d\} & g(ad) &= \{b, c\} \\ g(abc) &= \{d\} & g(abd) &= \{c\} & g(acb) &= \{d\} \\ g(acd) &= \{b\} & g(adb) &= \{c\} & g(adc) &= \{b\} \\ g(e_0 \dots e_3) &= \epsilon & & & & \end{aligned}$$

[3] to remove symmetries.

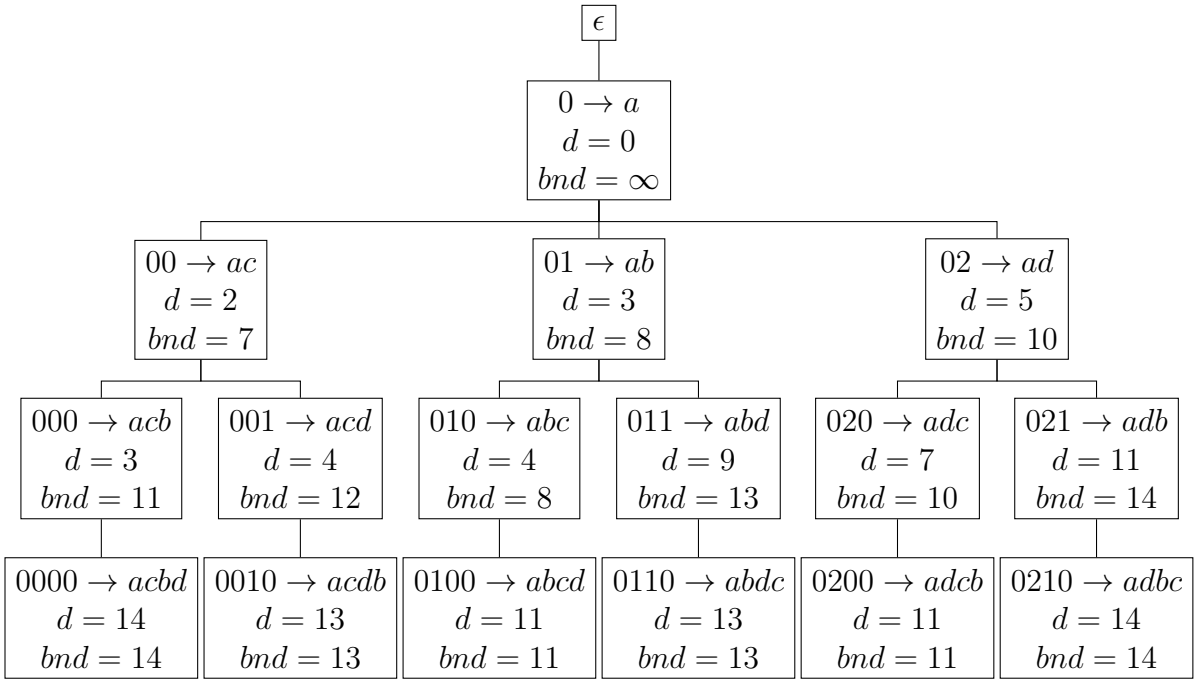


Figure 3.4: Search tree corresponding to the TSP instance of Figure 3.3.

Such that the final tree consists of the nodes $\{a, ab, acad, abc, abd, acb, acd, adb, adc, abcd, abdc, acbd, acdb, adbc, adcb\}$

One possible search heuristic is to explore children in order of increasing distance cost. To this end, we define an ordered tree generator, $h : \mathcal{C}^* \to \mathcal{C}^*$ as:

$$h(a_0 \dots a_k) = b_0 \dots b_i \text{ where } \{b_0, \dots, b_i\} = \mathcal{C} \setminus \{a_0, \dots, a_k\}$$
$$\text{and } \forall l, j \; l < j \leq i, d(a_k, b_l) \leq d(a_k, b_j)$$

That is, the children are any city not yet in the tour ordered in increasing distance from the last city chosen in the tour.

The complete search tree for the TSP instance given in Figure 3.3 is shown in Figure 3.4. Each node shows the ordered tree mapping (λ) from a tree over $\mathbb{N}$ to one over $\mathcal{C}$, the distance of the (partial) tour, and an upper bound based on a maximum spanning tree function (described in Section 5.2.3.2). The tree generator does not remove symmetries, i.e. $abcd = reverse(adcb)$, causing all solutions (including the optimal) to appear twice. An improved generator could account for this to reduce the search space further.

3.3 Semantics of Parallel Tree Search

Using the definitions of ordered trees, we construct a set of small-step reduction rules to model multi-threaded tree traversals. We assume an ordered tree λ over X that will be traversed

Traversal $\mathcal{T}$ (Section 3.3.4)	**Pruning $\mathcal{P}$ (Section 3.3.7)**
schedule	prune-decide
terminate	prune-optimise
advance	
Node Processing $\mathcal{N}$ (Section 3.3.6)	**Spawn $\mathcal{S}$ (Section 3.3.8)**
enumerate	spawn
decide	spawn-depth-bounded (Section 4.3.4.1)
optimise	spawn-stack-stealing (Section 4.3.5.1)
	spawn-budget (Section 4.3.6.1)

Table 3.1: MT^3 rule categories.

according to the order $\leq_{\text{lex}}$. Importantly the reduction rules only work when using an ordered tree as a total ordering of nodes is required[4].

The reduction rules specify the behaviour of parallel tree traversal by defining transitions (the rules) between program *states* (Section 3.3.3). Repeated application of rules correspond to the execution (on an abstract machine) of parallel tree traversal.

A summary of all the rules given in this section, and the spawn rules for each of the skeletons of Chapter 4, is given in Appendix A. We show the rules can successfully perform tree search by providing a Haskell program in Appendix B that implements the reduction rules for branch and bound optimisation searches.

3.3.1 Creating an MT^3 Model

MT^3 is a family of search models. The model for a specific search is created by choosing an appropriate set of reduction e.g. rules for a branch and bound optimisation problem.

The rules fit into the four categories shown in Table 3.1. The *Traversal* rules must be included in any model as these define the core tree traversal semantics. Additional rules are then chosen on a per search basis. One *Node Processing* rule must be chosen, e.g. enumerate, to determine the correct global state σ. *Pruning* and *Spawn* rules are both optional. Adding a *Pruning* rule enables branch and bound support while *Spawn* rules allow for parallelism.

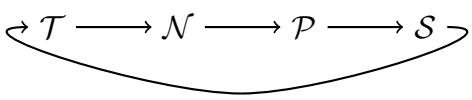
The skeleton designs in Chapter 4 mimic the separation of core tree traversal rules and additional search-specific functionality. As the skeletons affect only *Traversal* (to allow different workqueue policies on a schedule) and *Spawn* rules; the same parallelisation can be applied to all search types, increasing reusability.

[4]Any unordered tree can be converted to an ordered tree by simply enforcing some ordering on nodes; even if there is no search heuristic underpinning this.

3.3.2 Rule Ordering

To ensure correctness, we must constrain the order that the categories of rule are applied. This avoids, for example, not processing a node due to traversing twice without a process.

The ordering of rules depends on the category of the rule given in Table 3.1 (e.g. Traversal, Node Processing etc.). Rules should be applied as follows:



That is, if the last rule we applied was a Traversal ($\mathcal{T}$) then we should next attempt to apply a Node Processing ($\mathcal{N}$) rule. If no rule in a category applies then we try a rule from the next category. This ordering occurs on a per-thread basis rather than the entire search performing a single category of rules.

As Prune ($\mathcal{P}$) rules are only an optimisation it is allowable to skip this category. However, in this work, and practically, prune rules should almost always be applied where possible.

3.3.3 Search State

Let $n \geq 1$ be the number of threads. The *state* of a backtracking tree traversal takes the form $\langle \sigma, \mathit{Tasks}, \theta_1, \ldots, \theta_n \rangle$ where:

- σ stores global search state, e.g. best solution so far. The global search state depends on the type of search being performed and is specified in more detail in Section 3.3.6.
- Tasks is a queue of pending tasks, where a task corresponds to a *sub-tree* to be traversed. We use $[\,]$ to denote the empty queue, and $S{:}\,\mathit{Tasks}$ to denote a non-empty queue with head S, where S is a sub-tree. The notation $\mathit{Tasks}{:}S$ represents adding S to the tail of Tasks.

  In practice the structure of Tasks and order of removal from Tasks is important for maintaining heuristic search orders, and is discussed in detail in section Section 4.4. Here we assume tasks are removed in the order they are inserted (first-in, first-out).
- θ_i is the state of the i^{th} thread. We use $\bot$ to represent an idle thread or $\langle S, v \rangle$ to represent a thread currently exploring node v in sub-tree S.

Search begins with all threads idle and Tasks containing the entire search tree, i.e. $\langle \sigma, S_{root}{:}[\,], \bot, \ldots, \bot \rangle$. In cases where static work generation (Section 2.4.1) is used, e.g. Chapter 7, the Tasks may begin populated.

Search ends when the Tasks queue is empty and all threads are idle, i.e. $\langle \sigma, [\,], \bot, \ldots, \bot \rangle$.

3.3.4 Traversal Rules

General backtracking tree traversal is defined by the following three reduction rules, where the subscript i on the rule represents the thread executing the rule:

$$(\text{advance}_i)\quad \frac{u = succ(S, v) \qquad u \neq \bot}{\langle \sigma, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \sigma, \mathit{Tasks}, \ldots, \langle S, u\rangle, \ldots\rangle}$$

$$(\text{terminate}_i)\quad \frac{succ(S, v) = \bot}{\langle \sigma, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \sigma, \mathit{Tasks}, \ldots, \bot, \ldots\rangle}$$

$$(\text{schedule}_i)\quad \frac{v = \text{root of } S}{\langle \sigma, S{:}\mathit{Tasks}, \ldots, \bot, \ldots\rangle \rightarrow \langle \sigma, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle}$$

The *advance* rule is central to tree traversal and forms the majority of search steps. Given a node in a sub-tree the advance rule expands the search by moving to the next, $\leq_{\text{lex}}$, vertex to be explored.

terminate allows a running thread to move to the idle state, $\bot$, if and only if the currently allocated sub-tree has been fully explored. It does not need to return any results as these are stored in σ.

schedule ensures that threads do not idle if there is work to be performed. All work is drawn from the global *Task* queue.

3.3.5 Example Reductions for TSP

Given these three rules, a parallel reduction for the TSP problem of Section 3.2 is shown in Figure 3.5.

We assume there are two threads in the system, t_1 and t_2, and that threads are scheduled round-robin starting from thread 1. As we have not yet introduced a spawn rule we further assume that initially $\mathit{Tasks} = \{S_{ac}, S_{ab}, S_{ad}\}$, i.e. all sub-trees rooted at depth 1 in the search tree of Figure 3.4, corresponding to the partial tours $\{ac, ab, ad\}$ respectively.

Figure 3.5 shows both threads are initially scheduled, thereafter most steps become *advance* as the search tree is traversed. Because the sub-trees have the same size both threads terminate together. Due to a lack of available tasks a single thread completes the rest of the traversal, i.e. load balance is poor.

Rule	State
	$\langle \sigma, [S_{ac}, S_{ab}, S_{ad}], \bot, \bot \rangle$
(schedule$_1$)	$\langle \sigma, [S_{ab}, S_{ad}], \langle S_{ac}, ac \rangle, \bot \rangle$
(schedule$_2$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, ac \rangle, \langle S_{ab}, ab \rangle \rangle$
(advance$_1$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, acb \rangle, \langle S_{ab}, ab \rangle \rangle$
(advance$_2$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, acb \rangle, \langle S_{ab}, abc \rangle \rangle$
(advance$_1$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abc \rangle \rangle$
(advance$_2$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abcd \rangle \rangle$
(advance$_1$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, acd \rangle, \langle S_{ab}, abcd \rangle \rangle$
(advance$_2$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, acd \rangle, \langle S_{ab}, abd \rangle \rangle$
(advance$_1$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, acdb \rangle, \langle S_{ab}, abd \rangle \rangle$
(advance$_2$)	$\langle \sigma, [S_{ad}], \langle S_{ac}, acdb \rangle, \langle S_{ab}, abdc \rangle \rangle$
(terminate$_1$)	$\langle \sigma, [S_{ad}], \bot, \langle S_{ab}, abdc \rangle \rangle$
(terminate$_2$)	$\langle \sigma, [S_{ad}], \bot, \bot \rangle$
(schedule$_1$)	$\langle \sigma, [\,], \langle S_{ad}, ad \rangle, \bot \rangle$
(advance$_1$)	$\langle \sigma, [\,], \langle S_{ad}, adc \rangle, \bot \rangle$
(advance$_1$)	$\langle \sigma, [\,], \langle S_{ad}, adcb \rangle, \bot \rangle$
(advance$_1$)	$\langle \sigma, [\,], \langle S_{ad}, adb \rangle, \bot \rangle$
(advance$_1$)	$\langle \sigma, [\,], \langle S_{ad}, adbc \rangle, \bot \rangle$
(terminate$_1$)	$\langle \sigma, [\,], \bot, \bot \rangle$

Figure 3.5: Example reductions for the TSP instance of Section 3.2. Backtracking only.

3.3.6 Node Processing Rules

At this stage MT^3 only specifies how the tree is traversed (in parallel), but no useful work is performed as we have not yet specified the type of search (e.g. enumeration, decision or optimisation). The type of the search affects not only the rules, but also the global state σ.

3.3.6.1 Enumeration

Enumeration searches are closely related to the tree traversal above in that the entire search space is always explored. For each node searched we wish to track some information. For example: counting the number of nodes at a particular depth, counting the number of leaf nodes, or storing a representation for each node.

Let $\sigma_{enumeration} = \{m\}_e$, where m is an element of a *monoid* $\langle M, id, + \rangle$. The e subscript on the state distinguishes this as an *enumeration* state. Processing a node corresponds to the following rule, where $h : X^* \to M$ maps search tree nodes into the monoid:

$$\text{(enumerate}_i\text{)} \quad \frac{}{\langle \{m\}_e, \mathit{Tasks}, \ldots, \langle S, v \rangle, \ldots \rangle \to \langle \{m + h(v)\}_e, \mathit{Tasks}, \ldots, \langle S, v \rangle, \ldots \rangle}$$

By using a monoid we abstract over the details of what is being enumerated. This improves the generality of the enumeration search. For example, in the case studies presented in Section 5.2.1 the monoid always takes the form of a map between depth and vertex count, where $m + h(v)$ adds one to the map at the depth of v. To count the leaf nodes the monoid would treat a $h(v)$ as id if it is a non-leaf and add one to a count if it is a leaf.

For correctness the *enumerate* rule must fire **exactly once** for each node in the search tree. This corresponds to firing after each advance and schedule rule (on the same thread).

3.3.6.2 Decision

Decision problems search for a node with a particular property known as the *target node*. If a target node is found termination can occur before exploring the entire search tree.

Let $\sigma_{decision} = \{v\}_d$ if a target node v is found, otherwise $\{\}_d$. A function $match : X^* \to \{true, false\}$ returns true if a node in the search tree matches the target node, otherwise t returns false.

Processing a node corresponds to the following rule.

$$(\text{decide}_i)\ \frac{match(v)}{\langle \{\}_d, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle \to \langle \{v\}_d, [\,], \bot, \ldots, \bot\rangle}$$

decide handles the early termination by putting the search into the final state $\langle \sigma, [\,], \bot, \ldots, \bot\rangle$ if it finds a target node.[5]

As with enumerate, *decide* must be applied if possible after every advance and schedule rule, to ensure a target node is never missed.

Using the TSP instance of Section 3.2 we show in Figure 3.6 how the reductions change to support decision searches. Here we assume work is statically generated by splitting into three sub-trees S_{ac},S_{ab} and S_{ac}. Let the target function $match$ return true if and only if a tour of length 13 is found. An additional tour length column shows the tour length of the node each thread is currently considering where ∞ represents a partial tour length and $\bot$ represents no node is currently being considered.

Due to early termination, the decision version of the search requires fewer reductions complete than the full traversal (Figure 3.5). As the search was found to be satisfiable no explicit *termination* rules were required as termination was handled by *decide*. Here we see the power of the early termination rule that allows the entire sub-tree S_{ad} to never be scheduled or traversed.

[5]The decide rule could alternatively only update σ and allow a specialised pruning rule to handle early termination.

Rule	State	Tour Length
	$\langle \{\}_d, [S_{ac}, S_{ab}, S_{ad}], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$
(schedule$_1$)	$\langle \{\}_d, [S_{ab}, S_{ad}], \langle S_{ac}, ac \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$
(schedule$_2$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, ac \rangle, \langle S_{ab}, ab \rangle \rangle$	$\langle \infty, \infty \rangle$
(advance$_1$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, acb \rangle, \langle S_{ab}, ab \rangle \rangle$	$\langle \infty, \infty \rangle$
(advance$_2$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, acb \rangle, \langle S_{ab}, abc \rangle \rangle$	$\langle \infty, \infty \rangle$
(advance$_1$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abc \rangle \rangle$	$\langle 14, \infty \rangle$
(advance$_2$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle 14, 11 \rangle$
(advance$_1$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, acd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle \infty, 11 \rangle$
(advance$_2$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, acd \rangle, \langle S_{ab}, abd \rangle \rangle$	$\langle \infty, \infty \rangle$
(advance$_1$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, acdb \rangle, \langle S_{ab}, abd \rangle \rangle$	$\langle 13, \infty \rangle$
(advance$_2$)	$\langle \{\}_d, [S_{ad}], \langle S_{ac}, acdb \rangle, \langle S_{ab}, abdc \rangle \rangle$	$\langle 13, 13 \rangle$
(decide$_1$)	$\langle \{acdb\}_d, [\,], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$

Figure 3.6: Example reductions for the TSP instance of Section 3.2. Decision variant.

Although two target nodes exist ($acdb$,$abdc$) the search only stores the first to be found[6]. In practice this can cause non-deterministic executions.

3.3.6.3 Optimisation

The final search type is optimisation, that searches for a node that maximises/minimises a particular objective function. Each node must be checked to determine if it improves the current *incumbent*. In Section 3.3.7 we show how pruning, based on the current incumbent, can be used to further reduce the search space.

Let $\sigma_{optimisation} = \{v\}_o$ where v is the current incumbent, initialised to the root node ϵ. A function $improves : X^* \times X^* \rightarrow \{true, false\}$ determines, for two search tree nodes u and v, if v is an improvement of the objective function compared to u. Processing a node corresponds to the following rule:

$$(\text{optimise}_i)\ \frac{improves(u, v)}{\langle \{u\}_o, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow \langle \{v\}_o, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle}$$

optimise must fire after each advance and schedule, if v improves u, to ensure no solution is missed.

Figure 3.7 shows how the reductions change to support optimisation search for the TSP instance of Section 3.2. As before, we assume work is statically generated by splitting into three sub-trees S_{ac},S_{ab} and S_{ac}. The function $improves(u, v)$ returns true if and only if the distance of the tour at node v is less than that of u (i.e. minimises the distance). For nodes

[6]The global state could be extended to store all solutions if required.

Rule	State	Tour Length	Incumbent Tour Length
	$\langle \{\epsilon\}_o, [S_{ac}, S_{ab}, S_{ad}], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	∞
(schedule$_1$)	$\langle \{\epsilon\}_o, [S_{ab}, S_{ad}], \langle S_{ac}, ac \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	∞
(schedule$_2$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, ac \rangle, \langle S_{ab}, ab \rangle \rangle$	$\langle \infty, \infty \rangle$	∞
(advance$_1$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, acb \rangle, \langle S_{ab}, ab \rangle \rangle$	$\langle \infty, \infty \rangle$	∞
(advance$_2$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, acb \rangle, \langle S_{ab}, abc \rangle \rangle$	$\langle \infty, \infty \rangle$	∞
(advance$_1$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abc \rangle \rangle$	$\langle 14, \infty \rangle$	∞
(advance$_2$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle 14, 11 \rangle$	∞
(optimise$_1$)	$\langle \{acbd\}_o, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle 14, 11 \rangle$	14
(optimise$_2$)	$\langle \{abcd\}_o, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle 14, 11 \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [S_{ad}], \langle S_{ac}, acd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle \infty, 11 \rangle$	11
(advance$_2$)	$\langle \{abcd\}_o, [S_{ad}], \langle S_{ac}, acd \rangle, \langle S_{ab}, abd \rangle \rangle$	$\langle \infty, \infty \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [S_{ad}], \langle S_{ac}, acdb \rangle, \langle S_{ab}, abd \rangle \rangle$	$\langle 13, \infty \rangle$	11
(advance$_2$)	$\langle \{abcd\}_o, [S_{ad}], \langle S_{ac}, acdb \rangle, \langle S_{ab}, abdc \rangle \rangle$	$\langle 13, 13 \rangle$	11
(terminate$_1$)	$\langle \{abcd\}_o, [S_{ad}], \bot, \langle S_{ab}, abdc \rangle \rangle$	$\langle \bot, 13 \rangle$	11
(terminate$_2$)	$\langle \{abcd\}_o, [S_{ad}], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	11
(schedule$_1$)	$\langle \{abcd\}_o, [\,], \langle S_{ad}, ad \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [\,], \langle S_{ad}, adc \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [\,], \langle S_{ad}, adcb \rangle, \bot \rangle$	$\langle 11, \bot \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [\,], \langle S_{ad}, adb \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [\,], \langle S_{ad}, adbc \rangle, \bot \rangle$	$\langle 14, \bot \rangle$	11
(terminate$_1$)	$\langle \{abcd\}_o, [\,], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	11

Figure 3.7: Example reductions for the TSP instance of Section 3.2. Optimisation variant.

with partial tours, $improves(u, v)$ always returns false. Because there is no pruning, the entire search space is explored to prove optimality of the solution, even though we find the minimum cost tour early in the search. As the search heuristics perform well we find the optimal solution quickly in the second thread.

3.3.7 Pruning Rules

An improvement for decision and optimisation searches is to use a bounding function to reduce the size of the search space by pruning sub-trees that provably cannot contain an optimal result or target node. That is, to move from fully backtracking to branch and bound search.

Let $p_d : X^* \to \{true, false\}$ be a pruning function for decision problems, where an upper bound of a node v is used to determine if the sub-tree rooted at v should be pruned, based on the target node. Let $p_o : X^* \times X^* \to \{true, false\}$ be a pruning function for optimisation problems, where given an incumbent node inc and a node v, $p_o(inc, v)$ determines if the

upper bound of v makes it impossible for the sub-tree rooted at v to contain a solution that improves the incumbent, i.e. should v be pruned?

We encode pruning using the following rules:

$$\text{(prune-decide}_i\text{)} \quad \frac{p_d(v) \qquad S' = subtree(S, v)}{\langle \{\epsilon\}_d, Tasks, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \{\epsilon\}_d, Tasks, \ldots, \langle (S \setminus S') \cup \{v\}, v\rangle, \ldots\rangle}$$

$$\text{(prune-optimise}_i\text{)} \quad \frac{p_o(u, v) \qquad S' = subtree(S, v)}{\langle \{u\}_o, Tasks, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \{u\}_o, Tasks, \ldots, \langle (S \setminus S') \cup \{v\}, v\rangle, \ldots\rangle}$$

Where pruning removes the sub-tree rooted at the current node, i.e. v. To ensure $succ$ is well defined, v must appear in both the sub-tree that has been removed and the tree it was removed from, i.e. $(S \setminus S') \cup \{v\}$.

For correctness the pruning rules should meet the following conditions, shown here for the maximising optimisation case (minimisation and decision cases follow similarly). We assume a function $obj : X^* \rightarrow \mathbb{R}$ that returns the objective value of a node, and a function $bnd : X^* \rightarrow \mathbb{R}$ returning an upper bound on the maximum objective possible for a given node, such that $\forall v, bnd(v) \geq obj(v)$.

1. $\forall u, v \in X^*$, if $p_o(u, v) = true$ then $obj(u) \geq bnd(v)$.
2. $\forall u, v, v' \in X^*$, if $v \preceq v'$ and $p_o(u, v) = true$ then $p_o(u, v') = true$
3. $\forall u, u', v \in X^*$, if $p_o(u', v) = true$ and $obj(u) \geq obj(u')$ then $p_o(u, v) = true$.

Condition 1 states that we should only ever prune a subtree that cannot possibly contain an improved solution. Condition 2 ensures that if the root of a subtree should be pruned, then any child node in the subtree should also be pruned. Condition 3 states that if a node v should be pruned with given an incumbent u', it should be pruned with any stronger incumbent u.

Conditions 2 and 3 are statements of the monotonicity of p_o. Non-monotonic reasoning rules exist [107] that allow, for example, randomisation to occur in p_o so long as no valid solutions are ever removed. Such reasoning is outwith the scope of this thesis and we restrict ourselves to the conditions given above.

Importantly, pruning rules are an optimisation only and not applying the rule, or applying it only in some cases, does not affect the correctness of search. Informally, we can show pruning does not affect correctness of search as follows, assuming a maximising optimisation problem[7].

[7]The decision variant follows in a similar fashion

Rule	State	Tour Length	Bounds	Incumbent Tour Length
	$\langle \{\epsilon\}_o, [S_{ac}, S_{ab}, S_{ad}], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	∞
(schedule$_1$)	$\langle \{\epsilon\}_o, [S_{ab}, S_{ad}], \langle S_{ac}, ac \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	$\langle 7, \bot \rangle$	∞
(schedule$_2$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, ac \rangle, \langle S_{ab}, ab \rangle \rangle$	$\langle \infty, \infty \rangle$	$\langle 7, 8 \rangle$	∞
(advance$_1$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, acb \rangle, \langle S_{ab}, ab \rangle \rangle$	$\langle \infty, \infty \rangle$	$\langle 11, 8 \rangle$	∞
(advance$_2$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, acb \rangle, \langle S_{ab}, abc \rangle \rangle$	$\langle \infty, \infty \rangle$	$\langle 11, 8 \rangle$	∞
(advance$_1$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abc \rangle \rangle$	$\langle 14, \infty \rangle$	$\langle 14, 8 \rangle$	∞
(advance$_2$)	$\langle \{\epsilon\}_o, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle 14, 11 \rangle$	$\langle 14, 11 \rangle$	∞
(optimise$_1$)	$\langle \{acbd\}_o, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle 14, 11 \rangle$	$\langle 14, 11 \rangle$	14
(optimise$_2$)	$\langle \{abcd\}_o, [S_{ad}], \langle S_{ac}, acbd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle 14, 11 \rangle$	$\langle 14, 11 \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [S_{ad}], \langle S_{ac}, acd \rangle, \langle S_{ab}, abcd \rangle \rangle$	$\langle \infty, 11 \rangle$	$\langle 12, 11 \rangle$	11
(advance$_2$)	$\langle \{abcd\}_o, [S_{ad}], \langle S_{ac}, acd \rangle, \langle S_{ab}, abd \rangle \rangle$	$\langle \infty, \infty \rangle$	$\langle 12, 9 \rangle$	11
(prune-optimise$_1$)	$\langle \{abcd\}_o, [S_{ad}], \langle (S_{ac} \setminus S_{acd}) \cup \{acd\}, acd \rangle, \langle S_{ab}, abd \rangle \rangle$	$\langle \infty, \infty \rangle$	$\langle 12, 9 \rangle$	11
(advance$_2$)	$\langle \{abcd\}_o, [S_{ad}], \langle (S_{ac} \setminus S_{acd}) \cup \{acd\}, acd \rangle, \langle S_{ab}, abdc \rangle \rangle$	$\langle \infty, 13 \rangle$	$\langle 12, 13 \rangle$	11
(terminate$_1$)	$\langle \{abcd\}_o, [S_{ad}], \bot, \langle S_{ab}, abdc \rangle \rangle$	$\langle \bot, 13 \rangle$	$\langle \bot, 13 \rangle$	11
(prune-optimise$_2$)	$\langle \{abcd\}_o, [S_{ad}], \bot, \langle (S_{ab} \setminus S_{abdc}) \cup \{abdc\}, abdc \rangle \rangle$	$\langle \bot, 13 \rangle$	$\langle \bot, 13 \rangle$	11
(schedule$_1$)	$\langle \{abcd\}_o, [], \langle S_{ad}, ad \rangle, \langle (S_{ab} \setminus S_{abdc}) \cup \{abdc\}, abdc \rangle \rangle$	$\langle \infty, 13 \rangle$	$\langle 10, 13 \rangle$	11
(terminate$_2$)	$\langle \{abcd\}_o, [], \langle S_{ad}, ad \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	$\langle 10, \bot \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [], \langle S_{ad}, adc \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	$\langle 10, \bot \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [], \langle S_{ad}, adcb \rangle, \bot \rangle$	$\langle 11, \bot \rangle$	$\langle 11, \bot \rangle$	11
(advance$_1$)	$\langle \{abcd\}_o, [], \langle S_{ad}, adb \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	$\langle 14, \bot \rangle$	11
(prune-optimise$_1$)	$\langle \{abcd\}_o, [], \langle (S_{ad} \setminus \{adb, adbc\}) \cup \{adb\}, adb \rangle, \bot \rangle$	$\langle \infty, \bot \rangle$	$\langle 14, \bot \rangle$	11
(terminate$_1$)	$\langle \{abcd\}_o, [], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	11

Figure 3.8: Example reductions for the TSP instance of Section 3.2. Branch and bound optimisation variant.

Assume pruning occurs in a state $\langle \{u\}_o, \mathit{Tasks}, \ldots, \langle S, v \rangle, \ldots \rangle$, that is, $p_o(u, v) = \mathit{true}$ implying $obj(u) \geq bnd(v)$ (by condition 1). The $improves : X^* \times X^* \to \{true, false\}$ function from the optimise rule ensures the objective value of the stored solution increases monotonically over time, such that at the end of search we have a solution node u' such that $obj(u') \geq obj(u)$. For correctness, $subtree(S, v)$ cannot contain any node w such that $obj(w) \geq obj(u')$. As $subtree(S, v)$ was pruned we know $obj(v) \leq obj(u')$, and, as we have a subtree, we know $\forall w \in subtree(S, v), v \preceq w$ therefore by condition 2 $\forall w \in subtree(S, v), obj(u') \geq obj(w)$ showing that pruning never affects the correctness of search. We further gain from condition 2 that, even if we do not apply the prune rule for $\langle S, v \rangle$, we can prune for any future node v' where $v \preceq v'$ without affecting the correctness of search.

Figure 3.8 shows how branch and bound affects the reductions for the TSP instance given in Section 3.2, where the bounds are calculated using a maximum spanning tree procedure (described in Section 5.2.2.1). In this case, prune-optimise fires twice when it determines the bound of the current node cannot possibly beat the incumbent, e.g. bound of 12 and incumbent of 11 in the first prune-optimise case. In this search pruning helps little as the pruning rules only remove a single node in each case. In practice instances are much larger and we expect to remove many more nodes.

3.3.8 Spawn Rules

So far we have assumed that the $Tasks$ workqueue is populated before search begins. To support dynamic parallelism (Section 2.4.2) we require a method to add new tasks to the workqueue during evaluation.

The following spawn rule allows sub-trees to be converted to tasks by adding them to the $Tasks$.

$$(\text{spawn}_i)\ \frac{u \in S \qquad v <_{\text{lex}} u \qquad S_u = subtree(S, u)}{\langle \sigma, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow \langle \sigma, Tasks{:}S_u, \ldots, \langle S \setminus S_u, v \rangle, \ldots \rangle}$$

Here *any* still-to-be explored node can be converted to a task by adding the sub-tree rooted by the node u to the task set.

This rule allows any unexplored node to be converted to a task. In reality search trees are created on-demand and only a sub-set of nodes will be available to spawn. In practice we want more control over what is spawned and the conditions governing when spawning occurs. The skeletons presented in Chapter 4 differ in how they manage spawns and more specialised spawn rules are given in Sections 4.3.4.1, 4.3.5.1 and 4.3.6.1.

As with pruning, not applying a spawn rule does not affect the correctness of search although it may affect the effectiveness of parallelism. Likewise spawning every node does not affect the correctness of search as only *future* nodes are spawned ensuring progress is made even if every node is converted to a task. A spawn rule may add multiple tasks to the workqueue in a single reduction, as in Section 4.3.4.

3.4 Lazy Node Generators: A Functional Interface for Tree Search

Conceptually MT^3 assumes that the search trees are fully materialised (i.e. in $\langle S, v \rangle$, S contains **all** nodes of the sub-tree) but this is inpractical. To do so would require large amounts of memory as, even though specific nodes are likely to require minimal storage, search trees often consist of millions of nodes. Furthermore it is not possible to determine all nodes in a tree *a priori* (if you could then search is unnecessary). In practice, instead of fully materialising the search tree, we alternate between tree construction and tree traversal.

Not fully materialising the tree is possible as:

1. Node Processing rules never access the current sub-tree S.

2. Pruning a sub-tree a from a partially generated set $\{a, b, c, \dots\}$ corresponds to removing a. Once a is removed children of a will never be generated.

3. *advance* only ever moves into a **direct** child, by generating the next level of the tree, or to a node at a lower depth. All nodes at a lower depth must already have been generated.

4. After a node and all its children have been explored we never explore the node again (due to $\leq_{\text{lex}}$ ordering) so do not need to store this.

The general spawn rule (Section 3.3.8) does not allow the tree to be partially generated as it must be able to convert any unexplored node to a task. As mentioned previously this rule is not practically implementable. To allow it to work with partially generated trees we can add a further constraint that spawn only converts a direct child or an (already generated) node at a lower depth. The skeleton spawn rules of Section 4.3.4, Section 4.3.6, and Section 4.3.5, apply this constraint.

3.4.1 Node Generators

A partial tree, e.g. $S = \{v, u, \dots\}$, is expanded only in the advance rule to find the next node to visit, i.e. $succ(S, v)$ may generate child nodes (or pick an already generate node). Children of a node v may be generated by applying the ordered tree generator function $g : X^* \to X^*$ to v and constructing a set of (ordered) child nodes. For example, $g(v) = acb$ gives $\langle va, vc, vb \rangle$ as the child nodes. There may not always be any children, i.e. when v is a leaf node $g(v) = \epsilon$. We use the term *Node Generator* to refer to a function that applies the tree generator g to a single node and returns the (ordered) set of child nodes. That is, Node Generators know how to both find and order the next node labels (as g does) **and** from these create the actual child nodes, i.e. from $g(v) = \{a\}$ create $\{va\}$. Node Generators are commonly known as branching functions/rules.

Depth-first tree search can be represented as a stack of Node Generators as illustrated in Figure 3.9.

As Generators are defined using an ordered tree generator function they implicitly encode ordering heuristics (Section 2.1.4.1) by creating child nodes in a left-to-right order.

3.4.2 Node Generators as a Programming Interface

Node Generators provide a general-purpose programming interface for tree search. As Node Generators are specific instantiations of ordered tree generators they, by extension, are general enough to encode any search tree application. By creating trees as stacks of Node Generators

Figure 3.9: Example tree search using a stack of Node Generators. $a \mid \{ab, ac\}$ may be read as ab and ac are generated from a.

(as in Figure 3.9) all MT^3 rules can be implemented in practice without knowledge of the domain-specific tree types. We exploit this fact in Chapter 4 to create domain-independent parallelisations of tree search that are parameterised by a user specific Node Generator.

From an implementation standpoint, a user can specify their search tree as a single function with the signature:

```
function generateChildren(NodeType n) → [NodeType]
```

This function not only specifies a suitable ordering of elements in the (ordered) list but also knows how to combine the parent node with child information to construct the child nodes. For example, the tree generator for TSP returns the next *city* (in a heuristic order) whereas the Node Generator encodes both the next cities **and** how this these are combined with the current tour (in an implementation defined manner).

In practice Node Generators often require additional information, for example in TSP the tree generator relies on access to the set of cities, C, and distance function d. In MT^3 these are implicitly accessible. In the programming interface we instead allow read-only search specific variable access through an abstract `SearchSpace` type. Node Generators therefore take the form of a function:

```
function generateChildren(SearchSpace space, NodeType n) → [NodeType]
```

As a practical example a Node Generator for TSP can be defined as follows:

```
1 function generateChildrenTSP(SearchSpace space, Tour n):
2   cs ← n.difference(space.cities)
3   cs.sort(λ x y → space.d(n.last(), x) <= space.d(n.last(), y))
4   children = []
5   for c in cs:
6     children.append([n ++ c])
7   return children
```

That is, given a tour n and the space containing the distance function d and city set $cities$, we first create the set of all unchosen cities (line 2) and sort them into increasing distance order from the last city included in the tour (line 3). To create search tree nodes, i.e. tours, we iterate over the (ordered) remaining cities and build the new tour by adding the city to the old (partial) tour (line 6).

Importantly, the Node Generator does not specify anything about *how* and *when* the search tree is constructed and this is handled transparently by the advance rule/skeleton implementations. The ability for a user to express a wide range of searches by implementing a single function for each is very powerful. Node Generators assume nothing about how the values are computed. For example, a user may perform domain-specific parallel node processing steps that are transparent to the calling code. Many of the case studies presented in 5.2 use this to perform data-parallel (vectorised) node processing.

In Section 5.1.3.5 we describe additional programming interface elements are required to, for example, provide bound information for nodes and pruning functions.

3.4.3 Lazy Node Generators

Node Generators create the full set of child nodes of a particular node v when they are created called, i.e. on an advance. This has two downsides:

1. The memory required to store the child nodes is proportional to the number of children. For large searches this burdens system memory. We also lose the benefit of depth-first search that memory requirements are low as only one node (and parents) are required at a time, i.e. only $max\_depth \times nodesize$ memory is require.

2. Sometimes the full set of child nodes are not required. For searches where the heuristic ordering is related to the bounding function it is often possible to remove all future child nodes "to-the-right" on a failed bound check. For example, maximum clique searches can use colouring to both determine an upper bound on the maximum size of a clique and also to determine a heuristic search order where the highest colour class is searched first. If the bound check fails, say with colour class five, then we can be sure all children to-the-right will likewise fail the bounding checks as they must have colour class less than or equal to five. We refer to this idea of pruning all tasks to-the-right as the *PruneLevel* optimisation.

These downsides can be overcome generating children *lazily*, that is, each child node constructed by the generator when *advance* asks for the next child. This mimics depth-first search implementations that usually operate in a loop, generating one child at a time. We use the term *Lazy Node Generator*[8] to refer to the interface of this form.

Laziness may be implemented by introducing Node Generator as an object with a next method that lazily returns the next child node, e.g.

```
1 class NodeGenerator {
2   NodeGenerator(SearchSpace space, NodeType n)
3   NodeType next()
4 }
```

It is always possible to move from a Lazy Node Generator to a strict Node Generator by having the `NodeGenerator` compute all child nodes when it is constructed and iterate over these values on a `next()` call.

[8]Lazy Node Generators are closely related to lazy list evaluation as well as the generators provided in languages such as python via the `yield` keyword.

Previous general-purpose search work adopts similar interfaces. Bobpp [75] allows the user to specify a `GenChild` class that is similar to the NodeGenerator above, however Bobpp inserts *all* children into a global priority queue (GPQ) when called. The interface of the MaLLBa skeleton library is similar [108] where the branch function inserts all children into a queue. The Muesli skeleton library features a branch function of the form `Node** branch(Node* n, int* size)` [109] where the user must explicitly allocate an result array and inform the system of the number of children through the size parameter. All these interfaces exclude lazy generation of nodes causing increased memory requirements and not allowing optimisations such as PruneLevel to be applied.

3.5 Summary

We introduce MT^3, a family of models based on small-step operational semantics, for describing parallel backtracking search. MT^3 builds directly on BBM [4] that likewise represents trees as partially ordered sets of words. Ordered tree generator functions (Section 3.1.1) allow us to describe both how to construct the search tree and ordering heuristics.

MT^3 is flexible enough to encode all three types of search (enumeration, decision and optimisation) as well as branch and bound variants. This flexibility is achieved by introducing 4 categories of rules. Traversal rules (Section 3.3.4) that specify thread scheduling, tree traversal, and thread termination. Node Processing rules that support specific search types (Section 3.3.1), i.e. enumeration, decision and optimisation (Section 3.3.6). Pruning rules for branch and bound search (Section 3.3.7). Finally, Spawn rules capture work generation (Section 3.3.8) allowing succinct descriptions of the search skeletons of Chapter 4.

In practice search trees are both generated and traversed at runtime. We have shown that the rules do not require fully generated tree structures and how the ordered tree generator functions allow us to partially generate sections of a tree as required. The notion of partially applying a (ordered) tree generator may be used to derive a suitable programming interface: Node Generators (Section 3.4). Lazy Node Generators are a memory reducing optimisation that allows a Node Generator to lazily return children (in a heuristic order). Lazy Node Generators provide a domain-independent interface suitable to provide user specific computation to the skeletons of Chapter 4. While similar programming interfaces exist, e.g. [75, 108, 109], they do not support laziness, nor are they derived in such a principled manner.

MT^3 is used in Chapter 4 to define an the features of an abstract task-parallel framework for tree search, to describe work generation conditions for the skeletons (Sections 4.3.4.1, 4.3.5.1 and 4.3.6.1) and to describe how performance anomalies can affect search (Section 7.2). The skeletons mimic the design of MT^3 by specifying a set of core *search coordinations* that

encompass Traversal and Spawn rules. These coordinations are then augmented with specific search type functionality e.g Node Processing and Pruning rules.

Chapter 4

Search Skeletons

This chapter describes a set of general-purpose algorithmic skeletons for search. The skeletons are parameterised by the Lazy Node Generators of Section 3.4 that provide a uniform method for specifying backtracking search trees. By separation of user specific code and the parallel coordination we gain interchangeable parallel implementations of search.

Section 4.1 describes the appropriateness and construction of skeletons for search. Search skeletons consist of two parts 1) a parallel search coordination that determines how tasks are stored, generated and load balanced, and 2) search type specific functionality covering enumeration, decision and optimisation searches as well as branch and bound variants. This approach mimics MT^3 (Chapter 3) that separates tree *Traversal* and *Spawning* rules–describing the search coordination–from *Node Processing* and *Pruning* rules that describe search specific functions. A search application is created by providing a skeleton a Lazy Node Generator describing how a specific search tree is built.

For performance portability the parallel search coordinations are designed against an abstract parallel framework (TSF), described originally in Archibald et al. [5], and in more detail in Section 4.2. Key features of TSF are asynchronous task-parallelism and distributed-memory support allowing scalability to take advantage of multi-core and cluster architectures, and, in the future, HPC setups. Issues with using deque-based work-stealing (Section 2.2.3.1) to manage tasks in search frameworks are discussed in Section 4.4, leading to the creation of a depth-pool structure that aims to maintain search heuristics (as much as possible) during load-balancing.

Three parallel search coordinations are discussed: Depth-Bounded (Section 4.3.4), Stack-Stealing (Section 4.3.5), and Budget (Section 4.3.6). Each differs in how they create and manage tasks. A Sequential search coordination is also available (Section 4.3.3) to aid debugging and to determine overheads of the skeleton approach compared to a hand coded sequential search (Section 6.3). Throughout this chapter we use the term coordination to mean search coordination.

The coordinations are general-purpose, each supporting all of enumeration, decision, and optimisation searches (Section 4.5). As a skeleton is formed of a search coordination and a search type this leads to 12 concrete search skeletons.

An implementation of the skeletons is described in Chapter 5 and evaluated in Chapter 6. In Chapter 7 we present a specialised search coordination that allows improved reasoning about parallel performance for branch and bound decision and optimisation problems.

4.1 Skeletons as a High-Level Parallelism Approach

Parallel tree search is often implemented on a per-application and per-scale basis, often requiring intrusive re-writes of existing sequential searches, and often without parallel code reuse in mind. A high-level parallel programming model for parallel tree search offers many benefits. It allows search domain experts access to parallelism without dealing with low-level implementation details and inversely allows parallelism researchers access to a wide range of search applications to develop both new search parallelisations and general-purpose techniques for irregular applications. By hiding low-level implementation details we gain performance portability over a range of architectures, i.e. the same code can run on both multi-core shared-memory machines and distributed-memory clusters without changes to the user code[1].

Parallel algorithmic skeletons, described in Section 2.2.4.1, provide abstractions of common, reusable, computational patterns, by separating domain-specific computation from parallel *coordination* features (i.e. details of the parallelism). As tree search, both backtracking and branch and bound, forms an abstract algorithmic pattern, it lends itself well to the skeleton model.

There is a risk that general-purpose solutions are slower than their hand-coded equivalents due to, for example, less domain optimisations and polymorphism. We show in Section 6.3 that, by using programming techniques such as template metaprogramming (e.g. [104]), overheads of the skeleton approach can be small (quantified as on average 6.1% slower than hand coded searches in Section 6.3). The reusable nature of skeletons has proved highly advantageous, allowing seven different search applications, of three different search types, to share the same parallel search coordinations (Section 6.8). Section 6.9 provides evidence that skeletonised searches can scale on up to 255 workers.

[1] Users may be required to provide serialisation instance for their node types to support message passing. Adopting parallelism based on recompute, rather than sending nodes, can remove this limitation at the cost of increased complexity to track recompute paths and time to find recompute the starting node.

4.1.1 Creating Search Skeletons

Skeletons have both an abstract meaning and a concrete parallel coordination behaviour [110]. The meaning determines the algorithm that is being abstracted over, e.g. a `map`, while the behaviour describes *how* the parallelism should take place, e.g. multi-core divide and conquer or data-parallelism on GPUs. Parallel behaviours may be given further parameters to control, for example, data partitioning e.g. into particular chunk sizes. This additional information does not affect the semantic meaning of the skeleton. Skeletons are then parameterised by a specific user computations to form (part of) an application.

As shown by MT^3, semantics of search are specified in two parts: the core *Traversal* rules that, in our case, provide depth-first search; and search type information, i.e. *Node Processing* and *Pruning*, that determine the specific type of search. A search skeleton without a search type is semantically valid but performs no useful work. Given this search skeletons must be constructed from both a search coordination, e.g. Depth-Bounded (Section 4.3.4), and a search type, e.g Decision. We adopt a skeleton naming scheme of *search coordination* + *search type* for example, a *DepthBoundedDecision* skeleton.

As with other skeletons, search skeletons are further parameterised by a user computation to create an application. For search, this takes the form of a Lazy Node Generator (Section 3.4) that encodes a specific search tree. The components that construct a search skeletons is shown graphically in Figure 4.1. The skeletons are extensible, allowing new search coordination methods to be created. For example a search coordination may provide best-first search or random creation of tasks.

The main use case for the search skeletons is to solve a single search instance rather than as a building block within a larger parallel programs, e.g. combining a map and reduce skeletons. As such, the skeletons are not designed with composition in mind, i.e. we do not support nesting a search within a search. Supporting nested search is both difficult, due to both sharing of workpools and the requirement to maintain two heuristic orders, nor have we seen a use-case where this is necessary. Multiple search skeletons may be run *in sequence* within a single program.

4.2 An Abstract Parallel Framework for Tree Search

The search coordinations are designed against an abstract distributed task-parallelism framework, the *tree search framework (TSF)*; first introduced in Archibald et al. [5]. By designing against TSF, the skeletons are not only search application independent but also framework/language independent. Distributed-memory support aids performance portability by allowing the framework to scale across to networks of localities as is common in modern high performance

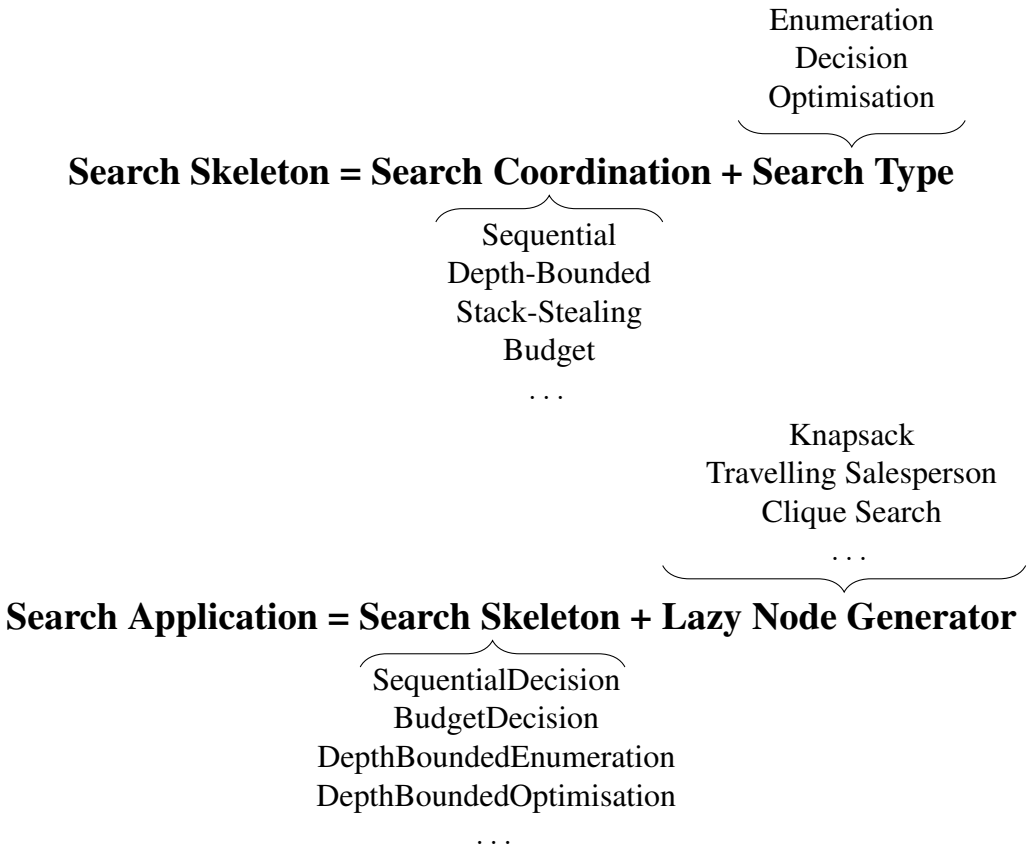


Figure 4.1: Parameters required to create a search skeletons and applications.

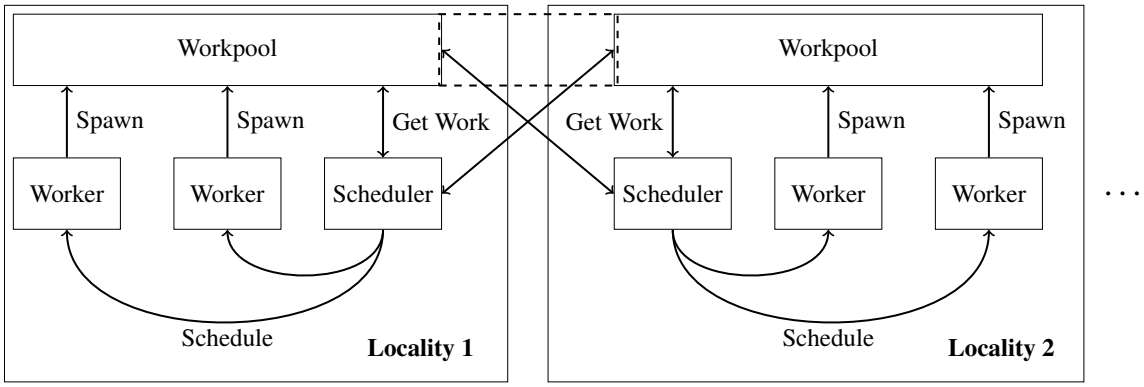


Figure 4.2: TSF: An abstract parallel framework for tree search.

environments. Section 5.1.1 and Section 5.1.2 show the benefits of designing to an TSF by describing the implementation of (a subset of) the skeletons in two different programming languages/parallel libraries.

TSF is shown in Figure 4.2. It consists of multiple localities (two shown) each with a set of workers and a single scheduler. Workers, when given a sub-tree, perform depth-first search. The scheduler is responsible for ensuring the workers are kept busy by either scheduling an existing task from the locality or finding tasks from another locality. The abstract framework assumes there are workpools but makes no assumptions as to their layout. For example we may have one workpool per-locality or a shared global workpool (as shown by the dashed lines between the workpools).

TSF closely resembles distributed task-parallelism frameworks such as X10 [111] or Chapel [112]. Likewise the definition of workpools as being either distributed or globally accessible (or something in between), mimics the separation provided by the global priority queue of Bob/Bobpp/Ibobpp [74, 75, 76].

The key features of TSF are as follows:

Distributed-memory support: As shown by MT^3 supporting any number of threads, we could treat TSF as a shared-memory architecture with infinite workers, however, given the differences in communication cost between localities (compared to within a locality) we make distributed-memory support explicit. This allows, for example, varying behaviour between load-balancing for local and remote tasks.

Asynchronous Task-Parallelism To support dynamic parallelism approaches (e.g. Section 2.4.2), new sub-tree search tasks can be created, via a *spawn* operation, dynamically at runtime. Newly spawned tasks are buffered in a workpool structure until a free worker is available. The choice of concrete workpool is non-trivial and is discussed in Section 4.4. Tasks themselves run completely independently, just as threads do in MT^3.

Distributed Scheduling Tasks are able to migrate between distributed-memory localities to support load-balancing. Schedulers ensure that workers are not idle if a local task is ready to execute, or dynamically search for work otherwise. When more than one task is available the tasks should be executed in an order specified by the workpool and search coordination. There is no requirement that the scheduler only interacts with a workpool structure when looking for work, e.g. direct communication between workers is possible. This moves away from the fixed workqueue model of MT^3 to allow more flexibility in the coordinations. As we will see in Sections 4.3.4.1, 4.3.5.1 and 4.3.6.1 the search coordinations may each be approximately described by the single workqueue model of MT^3.

Global Data Movement A key requirement of MT^3 is the ability to capture global state σ (Section 3.3.3). As such, the framework must support a method of data transfer, either with explicit messages or via a global address space. Global data movement allows the results of the searches to be shared/combined globally, for example when maintaining a global incumbent.

As search solutions are captured globally[2], there is no strict requirement for the framework to provide inter-task communication such as futures and promises [29] or direct message passing. In a concrete implementation (e.g. Section 5.1.3) inter-task communication is beneficial to support, for example, termination.

As TSF is a distributed framework it requires search applications to have serialisable node types to allow sub-trees to be transferred between (distributed) workers. An alternative approach, useful when dealing with search tree nodes that are costly to serialise/communicate, is to allow recomputing based on a path within a deterministic tree [113, 102]. We do not support recompute based approaches in this work.

4.3 Search Coordination Methods

Using TSF, we detail key the design choices for parallel search and introduce four general-purpose search coordination methods, Sequential, Depth-Bounded, Stack-Stealing, and Budget. Chapter 7 presents an additional skeleton implementation designed specifically for replicable branch and bound searches.

The search coordinations are given as a simple depth-first backtracking tree traversals. In Section 4.5 we show how search is specialised to a specific search type. All search coordinations introduce parallelism using space-splitting approaches (Section 2.4) with parallel tasks corresponding to sub-tree search.

[2]This feature distinguishes tree search from divide-and-conquer where results are combined locally.

4.3.1 Design Choices

The search coordinations are inspired by previous parallel search implementations. We can distinguish between parallel approaches based on:

1. **How** tasks are generated.
2. **When** tasks are generated.
3. **How** load-balancing occurs.

As described in Section 2.4 work may be generated either **statically** or **dynamically**. **Static** approaches operate on a fixed set of tasks that are often, but not always, generated in a work-generation phase before search. **Dynamic** approaches instead create tasks at runtime, often utilising some form of work-stealing model.

The search coordinations in this work rely on (distributed) work-stealing (Section 2.2.3.1) as the main mechanism for load-balancing, as is reflected by TSF.

4.3.1.1 Parallelisation Principles

Although **any** node in the search tree may be converted to a task, the skeletons are designed to choose heuristically *good* search nodes based on the following:

1. We should choose search nodes that will manifest large sub-trees in order to minimise scheduling overheads, i.e. we want to avoid scheduling many small tasks.
2. Nodes should be prioritised based on the applications' branching heuristics to quickly quite the search to promising nodes.

We expect the largest tasks to be those closest to the root as a) The amount of work in a task t corresponds to the number of nodes in the sub-tree rooted at t, and b) as we move deeper in the tree the search space becomes more constrained/smaller (for *finite* search spaces).

Lazy Node Generators implicitly encode branching heuristics in the order that child nodes are generated (Section 3.4). We should therefore aim to convert nodes to tasks in the order they come out of a generator.

Taken together, the guiding principle is that we should: **aim to create tasks from nodes as close to the root and as far to the left of the search tree as possible**. This principle has likewise been followed in previous work, e.g. [67].

Not all search coordinations strictly follow this principle, for example, we purposefully go against the heuristics when performing discrepancy search (Section 2.1.4.1) in the Ordered coordination described in Chapter 7.

While we adopt a steal left and close to the root as our guiding principle, other approaches are possible. For example, confidence based work-stealing [101] suggests that stealing low and left improves performance in some cases, although care must be taken to avoid communication overheads of scheduling small tasks.

It is often possible to assign some workers to search low in the tree and others to search high first. For example, by using a deque Section 4.4 to store tasks and allowing local workers steal the newest tasks (i.e. those that are lower in the tree) while remote steals receive older tasks. We adopt this approach in this thesis. Importantly, within a task we always generate work high and left first although the task itself may be searching a sub-tree low in the search tree.

4.3.2 Pseudocode

In the sections that follow we ue pseudocode to show the main operation of the search coordinations. The pseudocode has the following features:

- We assume a stack implementation exists with the usual `push` / `pop` / `empty` methods and a `top` method for inspecting the top node without removing it.
- `spawn` is a primitive that creates a new task from the given function and arguments. Importantly, it computes any arguments to the function (e.g. depth + 1) before generating the task, but does not run the given function.
- The Lazy Node Generator API has been extended to support a `hasNext` method, returning false if all children have been generated. This is for readability only and is not essential in the the implementations described in (Section 5.1).
- We support higher order functions.

4.3.3 Sequential Search Coordination

The simplest coordination is Sequential. Sequential corresponds to an MT^3 model with no *Spawn* rule, i.e. it does depth-first search from the root node. It is a static approach that generates a single task; searching the root node.

Pseudocode of a stack based implementation of Sequential, featuring the Lazy Node Generator API (Section 3.4), is in Listing 4.1. It is also possible to implement Sequential recursively.

Listing 4.1: Pseudocode for the Sequential search coordination.

```
1  function sequentialSearch(SearchSpace space, Node root):
2    generator ← nodeGenerator(space, root)
3    generatorStack.push(generator)
4
5    while not generatorStack.empty() do
6      generator ← generatorStack.top()
7      if generator.hasNext() then
8        node ← generator.next()
9
10       // Search type specific node processing is added here
11       // This will possibly force a continue instead pushing children to the stack,
12       // e.g. on a prune
13
14       generatorStack.push(NodeGenerator(space, node))
15     else
16       generatorStack.pop() // Backtrack
```

Sequential is useful for testing and debugging before adding parallelism. In Section 6.3 it is used to quantify the overheads of moving from hand written searches to those based on Lazy Node Generators.

4.3.4 Depth-Bounded Search Coordination

The Depth-Bounded coordination converts all nodes below a cut-off depth, d_{cutoff} into tasks. As d_{cutoff} is fixed, Depth-Bounded is a static parallelism approach.

Pseudocode for Depth-bounded is in Listing 4.2. As with Sequential, Depth-Bounded may be implemented recursively. From line 21 onward Depth-Bounded corresponds almost exactly to Sequential, the only difference being code to track the current depth (e.g. lines 28 and 30). The condition for spawning is expressed on line 6. Spawning occurs based on the child depth, i.e. `currentDepth` $+ 1$. An implementation of Depth-Bounded does not need to always check this condition as once the current depth is above the cutoff no more work will be generated by any children. Line 14 creates a new task and adds it to a local workpool.

A graphical depiction of Depth-Bounded is in Figure 4.3. This shows a simple scenario where all nodes at depth 1 are converted to tasks. After spawning all tasks, w_1 is left with no work and needs to go through a scheduling phase before continuing search. One optimisation, not used in this work, is to have a worker keep the leftmost task and spawn all other children to avoid the scheduling overhead.

Depth-Bounded has the advantage of being easy to implement, requiring few changes to a sequential implementation other than tracking the current depth (only needed when we are below d_{cutoff}) and the conditional to check if spawning should occur.

Although tasks are generated *statically*, based on d_{cutoff}, they are not all generated at the start of the search. Instead tasks spawns only occur when a node below d_{cutoff} is actually explored. This has two main effects. Firstly, spawns can take place on any locality in the

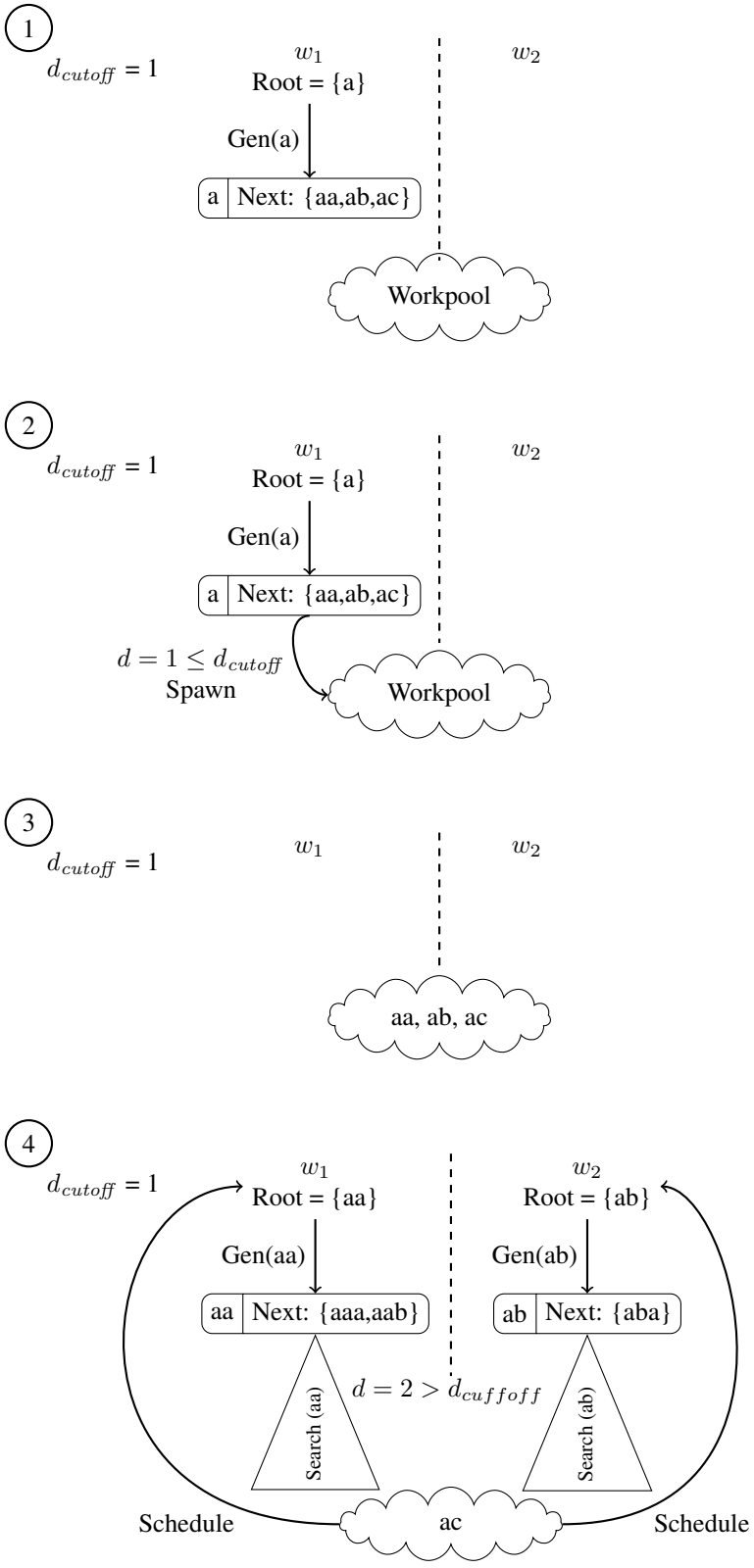


Figure 4.3: Operation of the Depth-Bounded search coordination. (1) An initial state with a worker w_1 having converted the root node a into a generator. (2) As $1 \leq d_{cutoff}$ a spawns take place, where 1 is the *child* depth. (3) Tasks waiting to be scheduled. (4) Two tasks are scheduled and expanded by the two workers. As the children of these nodes are at a depth 2 no further spawns occur.

Listing 4.2: Pseudocode for the Depth-Bounded search coordination.

```
function depthBoundedSearch(SearchSpace space, Node root, int cutoff, int currentDepth=0):
  generator ← nodeGenerator(space, root)
  generatorStack.push(generator)

  // Spawn condition
  if currentDepth + 1 <= cutoff then
    while generator.hasNext() do
      node ← generator.next()

      // Search type specific node processing is added here.
      // This will possibly force a continue instead of a spawn
      // e.g. on a prune

      spawn(depthBoundedSearch(space, node, cutoff, currentDepth + 1))
    return // root generator is exhausted. Exit.

  // Sequential Search
  while not generatorStack.empty() do
    generator ← generatorStack.top()
    if generator.hasNext() then
      node ← generator.next()

      // Search type specific node processing is added here.
      // This will possibly force a continue instead pushing children to the stack,
      // e.g. on a prune

      generatorStack.push(NodeGenerator(space, node))
      currentDepth ← currentDepth + 1
    else
      currentDepth ← currentDepth - 1
      generatorStack.pop() // Backtrack
```

system promoting a better load balance in distributed settings. Secondly, for searches that support pruning, spawning later in the search can reduce the total number of spawns as nodes are pruned before spawning occurs[3].

Spawned tasks are buffered in a workpool until they are requested by a scheduler. Depth-Bounded makes no assumptions on the type or locality of the workpool and it may be implemented as, for example, a single global workpool or using a workpool per-locality. The issue of workpool choice is explored further in Section 4.4.

The depth cutoff d_{cutoff} provides some control of task granularity and is commonly found in divide-and-conquer and fork-join applications. The idea is to split the search space into a number of large search tasks. Using d_{cutoff} avoids generating small tasks where parallel overheads dominate, i.e. the time to schedule a task is higher than the time to execute. For $d_{cutoff} = 0$ Depth-Bounded mimics Sequential, spawning the root task only. For $d_{cutoff} = \infty$ the skeleton mimics node-oriented search where tasks consists of taking a search tree node from a workpool, branching, pruning if required, and inserting all remaining children into the workpool (e.g. [85, 114]).

One disadvantage of Depth-Bounded is that d_{cutoff} must be tuned by the user on a per instance basis. While in practice we have found a d_{cutoff} of two levels below the root to perform

[3]This is the main reason pruning occurs before spawning in MT^3 (Section 3.3.2).

well for many instances this is not true in general (Section 6.5). Manually choosing d_{cutoff} also affects performance portability as it must account for the parallel architecture, i.e. more workers will require more tasks to keep them busy. Automatic tuning of cutoff depths remains an active research area [115] that is complicated further by the irregularity of search. Otten and Dechter [46] have considered d_{cutoff} tuning specifically for search, showing a dynamic cutoff depth scheme applied to a specific style of branch and bound search. It is not clear if these results generalise and we have not explored this here. We show the how the choice of d_{cutoff} affects performance in Section 6.5.

Existing static approaches (Section 2.4.1) provide alternatives to d_{cutoff}. For example, Embarrassingly Parallel Search [54] allows a user to specify a number of tasks per-worker. Unfortunately it is difficult to enable this in a distributed environment without generating all tasks upfront, potentially loosing the load balance and pruning benefits.

4.3.4.1 MT^3 Spawn Rule

Search coordinations determine how work is generated during a search. As such, they manifest themselves as *Spawn* rules in MT^3 (Section 3.3.1).

To express Depth-Bounded with MT^3 we assume that a single global workpool is used for all spawns. In practice distributed workpools are often used to promote improved load-balancing.

The following rule captures the work generation for Depth-Bounded:

$$\text{(spawn-depth-bounded}_i\text{)}\ \frac{|v|+1 \leq d_{cutoff} \qquad \{S_{c_1} \dots S_{c_n}\} = \{subtree(S,u) \mid u \in children(v)\}}{\langle \sigma, Tasks, \dots, \langle S, v\rangle, \dots\rangle \rightarrow \langle \sigma, Tasks{:}S_{c_1}\dots{:}S_{c_n}, \dots, \langle S \setminus S_{c_1} \setminus \dots \setminus S_{c_n}, v\rangle, \dots\rangle}$$

That is, if we are exploring a node v with children depth above the user specified d_{cutoff} we should convert each child of v to a task and add it to the global workpool.

We must spawn **all** child nodes in a single step to ensure correct rule ordering (Section 3.3.2). If not all children are spawned then the next traversal rule (*advance*) may enter a child node that should have been converted to a task.

4.3.5 Stack-Stealing Search Coordination

The Stack-Stealing coordination, like many dynamic approaches (Section 2.4.2), allows the search tree to be split on receipt of a work request rather than upfront. That is, Stack-Stealing allows work-stealing to happen directly from the generator stack of another worker. This contrasts the usual work-stealing approaches that steal existing tasks from workpools rather than causing work to be generated.

Listing 4.3: Pseudocode for the Stack-Stealing Search Coordination.

```
1  function stackStealingSearch(SearchSpace space, Node root, StealRequest stealRequest)
2    generator ← nodeGenerator(space, root)
3    generatorStack.push(generator)
4
5    while not generatorStack.empty() do
6      // Handle possible steals
7      if stealRequest then
8        // Find work from the top of the tree
9        for stealGen ← generatorStack.bottom() to generatorStack.top() do
10         if stealGen.hasNext() then
11           stealReq ← createNewStealRequest() // To allow steals on the new worker
12           respond(stealRequest.thief,
13                   stackStealingSearch(space, stealGen.next(), stealReq))
14           responded ← true
15           break
16
17       if not responded then
18         respond(stealRequest.thief, Failed) // Signal to the thief that stealing failed
19
20     // Continue sequential search
21     else
22       if generator.hasNext() then
23           node ← generator.next()
24
25           // Search type specific node processing is added here
26           // This will possibly force a continue instead pushing children to the stack,
27           // e.g. on a prune
28
29           generatorStack.push(NodeGenerator(space, node))
30       else
31           generatorStack.pop() // Backtrack
```

Pseudocode for Stack-Stealing is in Listing 4.3. Stack-Stealing must use a stack based implementation as a steal request requires full access to the generator stack in order to find the task that is leftmost and closest to the root, i.e. the task we expect to be most promising to steal (Section 4.3.1.1). Stack-Stealing checks the `stealRequest` on **every** search expansion step to determine if work is required (line 7). If a steal is requested then the generator stack is iterated through from bottom to top (line 9), i.e. nodes closest to the root first, until a generator with an unexplored child is found (line 10). If an unexplored child is found then a new task is given to thief (line 13). If no nodes are available to steal then the requested worker is notified (line 18).

An interesting feature of Stack-Stealing is that we do not require a workpool structure as workers communicate directly with each other. For simplicity we introduce a globally writable `StealRequest` type that allows a thief to signal when a steal is required. The `respond` function allows a node to be returned to the thief. The function `createStealRequest` hides the implementation of steal requests to aid readability.

In practice the implementation of `stealRequest` (including `respond` and `createStealRequest`) must be implemented in a thread safe manner, e.g. to avoid two workers stealing from a single worker at the same time. For simplicity we do not show thread saftey constructs here.

A graphical depiction of Stack-Stealing is in Figure 4.4. This shows how workers, on receipt of a work request, pause their current search to find the lowest, leftmost node and return it **directly** to the thief.

While not essential, the Stack-Stealing implementation work supports a mix of work-stealing and work-pushing (not shown in Listing 4.3). The search is initially split and eagerly scheduled such that there is one sub-tree per-worker. Once the initial task is complete idle workers resort to work-stealing. This improves load balance at the start of search by avoiding all workers trying to steal from the single (root node) worker.

Like other approaches based on work-stealing we use random victim selection, i.e. workers can steal from any other worker. In a distributed environment Stack-Stealing only steals from a remote locality when there are no active local workers[4], although this is not a requirement for correctness.

Unlike Depth-Bounded, Stack-Stealing does not require additional parameters from the user. This allows it to adapt both to instances and architectures thereby allowing performance portability and transparent scaling. In Section 6.8 we show Stack-Stealing, while not always being the best choice of skeleton for a given application, gives the best average performance over all applications/instances.

A disadvantage of Stack-Stealing is that work requests occur on the critical path of a worker that must 1) pause the current search 2) find a suitable node for the thief (if one exists) and 3) due to lazy generation, actually generate the node. If there are a large number of steals these overheads may dominate. We can alleviate this issue using *chunking*. Chunking allows workers to return multiple nodes on a steal request that can be buffered in a workpool for later use, i.e. a worker can take from the workpool instead of interrupting another worker. Due to the Lazy Generator Model, chunking increases the steal time as the search worker must call the generator n times. In Section 6.6.1 we show that chunking is often not beneficial in practice.

Currently a steal always succeeds if the victim has any unexplored tasks regardless of depth. This can cause many small tasks to be stolen increasing scheduling overheads. For example, if a worker, w_1, steals a node near a leaf of the tree and then another worker steals from w_1, the sub-tree stolen from w_1 will be even closer to a leaf node even though it is locally the most promising task. One way around this is to introduce a lower depth cut-off (as in [101]) that allows steals only occur if a worker has an unexplored node above this depth. Unfortunately this approach introduces an additional parameter for the user to specify, decreasing portability.

Stack-Stealing is influenced by existing dynamic approaches based on distributed work-stealing of tree nodes. Stack-Stealing resembles the approach of Abu-Khzam et al. [102]

[4]We define active workers as those that are currently searching **and** and are not currently being stolen from.

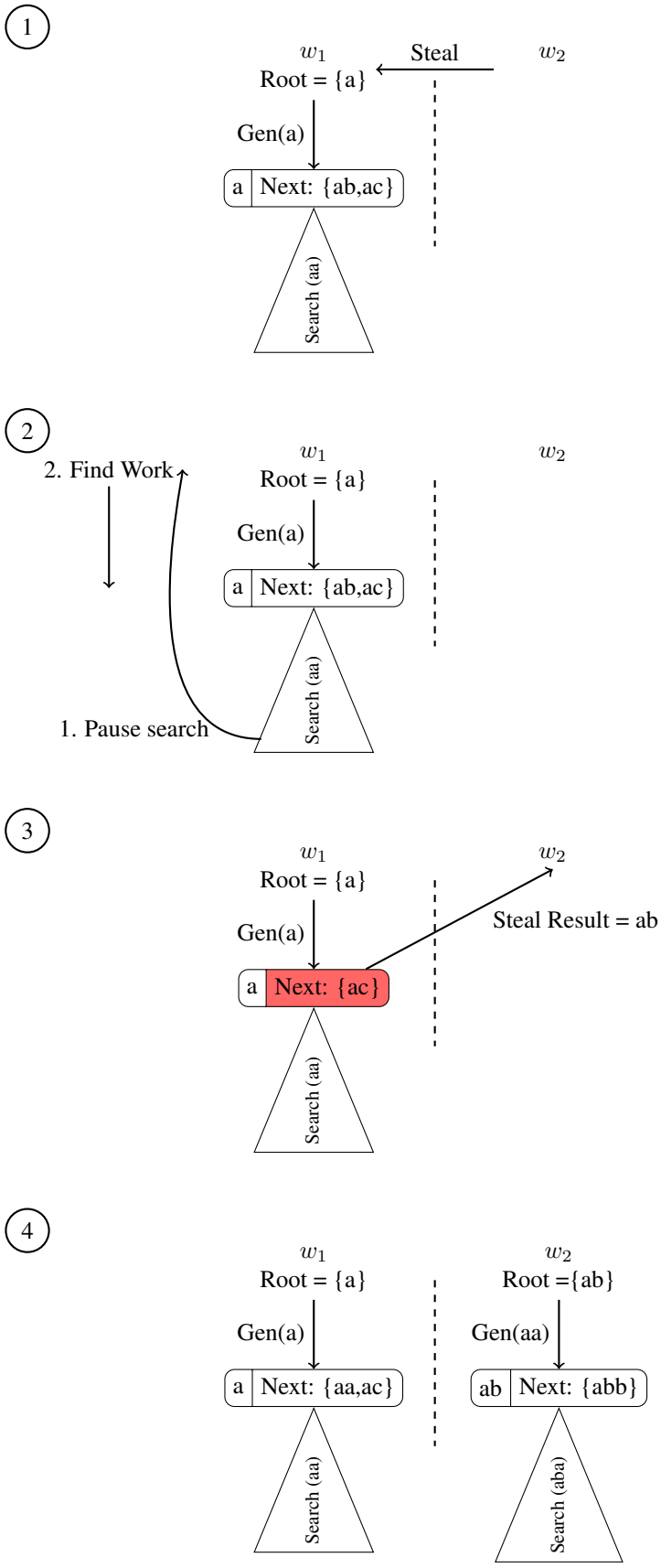


Figure 4.4: Operation of the Stack-Stealing search coordination. (1) A worker w_1 searching a tree rooted at a, while a second worker w_2 issues a steal request. (2) w_1 pauses search and starts back at the top of the stack looking for work (3) Work is found at the top level and $\{ab\}$ is sent to w_2. (4) w_1 resumes search while w_2 starts search from the stolen node $\{ab\}$.

that supports both random work-stealing of tasks from near the root first. However Stack-Stealing steals nodes directly rather than paths to nodes as in their approach. By following the parallelism principle of stealing high and left we ensure the search order distributed as little as possible. The approach of Abu-Khzam et al. performs steals high and to the right (breaking heuristic ordering).

4.3.5.1 MT^3 Spawn Rule

As with Depth-Bounded, we can describe the work generation behaviour of Stack-Stealing as an MT^3 spawn rule.

Stack-Stealing allows direct stealing of nodes between workers that is not supported by MT^3. Instead, we mimic Stack-Stealing behaviour by having the *victim* worker detect when there is an idle thread and spawn work to the workpool, allowing an idle thread to execute a schedule rule.

Spawning in Stack-Stealing (without chunking) is captured by the following rule:

$$\text{(spawn-stack-stealing}_i\text{)}\ \frac{u = nextLowest(S, v) \qquad u \neq \emptyset \qquad S_u = subtree(S, u)}{\langle \sigma, [\,], \bot, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \sigma, [S_u], \ldots, \langle S \setminus S_u, v\rangle, \ldots\rangle}$$

Where $nextLowest(S, v)$ gives the lowest, leftmost unexplored node, i.e. the most promising node to steal (Section 4.3.1); if there is an unexplored node present. $nextLowest(S, v)$ is defined as:

$$succ'(S, v) = \{u \in S \mid v \leq_{\text{lex}} u\}$$
$$nextLowest(S, v) = \{u \in succ'(S, v) \mid |u| = min_{||}(succ'(S, v) \wedge \forall w \in succ'(S, v),\ v \leq_{\text{lex}} u \leq_{\text{lex}} w\}$$

Where $min_{||}(S)$ returns the smallest cardinality in S, e.g. $min_{||}(\{00, 001, 012\}) = 2$.

Importantly the *spawn-stack-stealing* rule only applies when there is at least one idle thread and no tasks are waiting to be scheduled.

Chunking can be added by spawning the lowest level of children:

$$\{c_1 \ldots c_n\} = lowest(S, v) = \{u \in succ'(S, v) \mid |u| = min_{||}(succ'(S, v)\}$$

That is, all nodes at the lowest depth that have not yet been explored.

Each child $c_1 \ldots c_n$ is added to the workpool and removed from the current sub-tree S in a similar manner to Depth-Bounded (Section 4.3.4.1).

Listing 4.4: Pseudocode for the Budget search coordination

```
1  function budgetSearch(SearchSpace space, Node root, int backtrackThreshold):
2    backtracks ← 0
3    generator ← nodeGenerator(space, root)
4    generatorStack.push(generator)
5
6    while not generatorStack.empty() do
7    // Spawn condition
8      if backtracks >= backtrackTheshold then
9        // Find work from the bottom of the stack, i.e.\ closest to the root
10       for stealGen ← generatorStack.bottom() to generatorStack.top() do
11         if stealGen.hasNext() then
12           while stealGen.hasNext() do
13             node ← stealgen.next()
14             spawn(budgetSearch(space, node, backtrackThreshold))
15           break
16       backtracks ← 0
17
18     // Continue sequentially
19     else
20       if generator.hasNext() then
21           node ← generator.next()
22
23           // Search type specific node processing is added here
24           // This will possibly force a continue instead pushing children to the stack,
25           // e.g. on a prune
26
27           generatorStack.push(NodeGenerator(space, node))
28       else
29           backtracks ← backtracks + 1
30           generatorStack.pop() // Backtrack
```

4.3.6 Budget Search Coordination

The Budget coordination is influenced by existing periodic load-balancing approaches (Section 2.4.2.1). A period is defined on a per-worker basis, i.e. asynchronous periodic, based on the number of backtracks the worker has performed. Workers search sub-trees until the task is complete or the a user defined backtracking *budget* is met. When the budget is exhausted, all nodes at the lowest possible depth are spawned and the budget is reset.

Pseudocode for the Budget coordination is in Listing 4.4. As with Stack-Stealing, this must be written with an explicit stack to allow the task at the lowest depth to be spawned when the budget is exhausted.

Budget requires a simple change to the sequential portion of the search to keep track of the number of backtracks (line 29). For **every** node expansion step, Budget first checks that the backtrack budget has not been exhausted (line 8). If it has then the generator stack is iterated through, from the generator closest to the root, until a generator with children remaining is found. On finding a generator, budget spawns **all** of the child nodes of the generator (line 12) and resets the budget. If no unexplored nodes are available the budget resets to zero without spawning any new tasks.

A graphical depiction of Budget is in Figure 4.5 showing how Budget pauses search when the budget is exhausted in order to populate the workpool before restarting search.

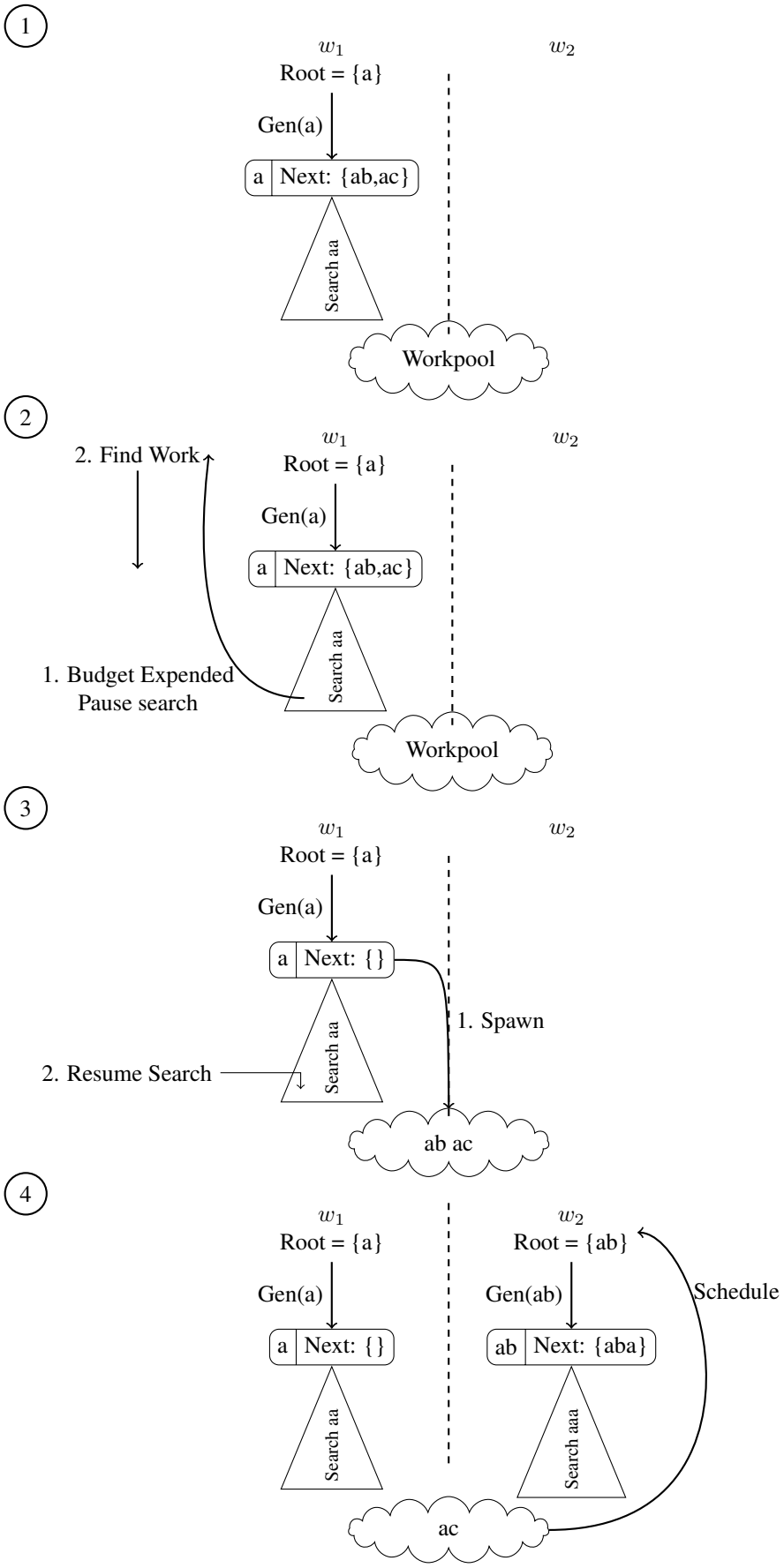


Figure 4.5: Operation of the Budget search coordination. (1) A worker w_1 performs a search rooted at a. (2) w_1 uses all its budget, pauses the search and retruns to the top of its stack to find work. (3) Work is found at the top level and spawned into the workpool. w_1 resumes search. (4) w_2 finds work in the workpool and begins to search from root ab.

The main principle behind Budget is that we should only spawn tasks we expect to be large. If a search finished before exhausting the budget then we conclude that there was a limited amount of work and it was not worth parallelising. If instead the budget is exhausted then we mobilise workers to help within this sub-tree.

We use number of backtracks as a budget that ensures search visits multiple branches in the tree before spawning. For searches with slow generator or node processing functions this may limit the amount of parallelism. Many other budget measures are possible such as: number of nodes expanded (used in mts [73]), number of valid solutions found, or time. It is not clear if there is a budget measure that is guaranteed to work well in all cases. In Section 6.7 we show that backtracks work well in practice.

The decision to spawn all top lowest depth tasks when the budget is exhausted is to ensure, particularly near the beginning of search, that enough work is created to keep workers busy. This choice is somewhat arbitrary, and, as with Stack-Stealing, we could spawn: a single task, n tasks, or even more than the lowest depth if required. We have not investigated these choices in this work.

As with Depth-Bounded, a major disadvantage of this approach is the requirement for the user to select a suitable budget. We show how the choice of budget affects search performance in Section 6.7. Surprisingly, we show a budget of 10^5 to be appropriate, but not necessarily the best choice, for many instances and applications.

Automatically determining a suitable budget remains an open problem. One possibility is to introduce a dynamic budget parameter that varies based on system properties. For example, we may want to spawn more tasks, even if they are smaller, when we have additional free workers.

As with Stack-Stealing, Budget necessarily increases the amount of computation on the critical search path in order to perform backtrack counting and comparison.

Budget closely resembles mts [73] that applies a similar budgeting approach where unexplored sub-trees are returned to a master pool when buget is expended. We do not assume any structure on the workpool(s), allowing fully distributed setups.

The approach also takes inspiration from confidence based work stealing [101] which attempts to assign more workers to areas of the search tree where solutions are expected. Instead of solution density we assign workers based on the backtracking budget aiming to let idle workers help with sub-trees we are confident can benefit from parallelism.

Search Coordination	Work Generation	Work Distribution
Sequential	Static, single root task	Assign to single thread
Depth-Bounded	Static, all tasks below depth d_{cutoff}	Work-stealing from workpools
Stack-Stealing	Dynamic, on work request	Work-stealing from workers
Budget	Dynamic, as budget is exhausted	Work-stealing from workpools
Ordered (Chapter 7)	Static, all tasks at depth d_{cutoff} (upfront)	Work-stealing from global fixed priority workpool

Table 4.1: Parallel search coordination work generation/distribution summary.

4.3.6.1 MT^3 Spawn Rule

The work generation behaviour of Budget, assuming a global workpool, can be captured as the following MT^3 spawn rule:

$$\text{(spawn-budget}_i\text{)}\ \frac{backtracks(i) = budget \quad \{c_1, \ldots, c_n\} = lowest(S, v) \quad \{S_{c_1}, \ldots, S_{c_n}\} = subtree(S, u)}{\langle \sigma, Tasks, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \sigma, Tasks{:}S_{c_1}{:}\ldots{:}S_{c_n}, \ldots, \langle S \setminus S_{c_1} \setminus \cdots \setminus S_{c_n}, v\rangle, \ldots\rangle \quad backtracks(i) = 0}$$

Where $lowest(S, v)$ returns the set of all vertices at the lowest depth in S as defined in Section 4.3.5.1.

The rule only applies when the number of backtracks performed by thread i has reached the $budget$ constant. All (unexplored) nodes at the lowest depth in S are spawned in a single operation.

We assume, for simplicity, that the function backtracks(i) returns the current number of backtracks for thread i. Setting backtracks(i) = 0 clears the backtrack count. This happens when a new task is started or when *spawn-budget* has been called[5].

4.3.7 Search Coordination Summary

The search coordinations determine the work generation and distribution of the parallel search skeletons, i.e. they encompass the *Traversal* and *Spawn* rules of MT^3. A comparison of the work generation and distribution choices for the search coordinations is in Table 4.1. The spawn rules of Sections 4.3.4.1, 4.3.5.1 and 4.3.6.1 allow us to succinctly describe the operation of the search coordinations[6].

The search coordinations are inspired by a range of existing approaches to parallel search (Section 2.4), including static approaches by Depth-Bounded, distributed dynamic work-stealing in Stack-stealing, and (asynchronous) periodic load-balancing in Budget. While

[5]The number of backtracks can be explicitly added to the rules by representing thread state as $\langle S, v, n\rangle$, where n is the number of backtracks, and updating n in the *advance*, *schedule* and *spawn* rules.

[6]The Ordered coordination cannot be fully described in MT^3 as discussed in Section 7.4.0.1.

these represent a mix of approaches, the nature of skeletons makes it easy to experiment with additional parallel search coordinations. For example, a dynamic centralised approach, or an approach based on the spawn rule of Section 3.3.8 where we decide at random if a new task should be created[7].

A recurring disadvantage is the tuning of skeleton parameters. Choosing appropriate parameters needs to account for both the instance being considered, e.g. high d_{cutoff} values are required for instances with low branching factors near the root, and also the parallel architectures, i.e. more workers require more tasks to be generated. Given the irregularly nature of search it is difficult to predict ahead of time the specific properties of an instance. Even if the properties of an instance are known it is not always clear how to translate these to appropriate parameters.

Requiring user-set parameters also leaks parallelism details to the skeleton user who must be aware of how d_{cutoff} or budgets are being used in order to choose appropriate values. This breaks the separation of computation and coordination. Ideally the skeletons should be able to operate independent of user input, as with Stack-Stealing, with the ability for expert users to give additional information for improved performance. This mirrors skeletons such as `map` that work without user input but also allow parameters such as chunk sizes to be specified. One possible method to achieve this, not explored in this work, is to dynamically adjust the skeleton parameters at runtime[8]. This has the effect of changing static approaches to dynamic approaches.

4.4 Workpool Design Choices

So far workpool choice has been considered in an abstract manner, stating only that it must provide a buffer for tasks and allow tasks to be removed and scheduled in some order. However, workpool choice is key to ensuring search heuristics are maintained. Failure to do so can lead to detrimental search anomalies (Section 2.3.2.1) where large amounts of additional work is being performed compared with a sequential search.

A popular workpool implementation for work-stealing scheduling is based on a *deque*. The key idea, popularised by Cilk, is that local accesses (for both spawn and steal) should occur at the front of the queue while remote accesses (for steals only) occur at the tail of the queue [116]. The intuition is that older tasks, those at the tail, are more likely to recursively generate work. This gives the remote worker a better chance of filling its own workpool; reducing the total overall steals. This idea is essential to modern work-stealing scheduling. Some

[7] Similar to the approach in PICO [86] that probabilistically releases nodes to a hub locality.
[8] This does not apply to the Ordered coordination of Chapter 7.

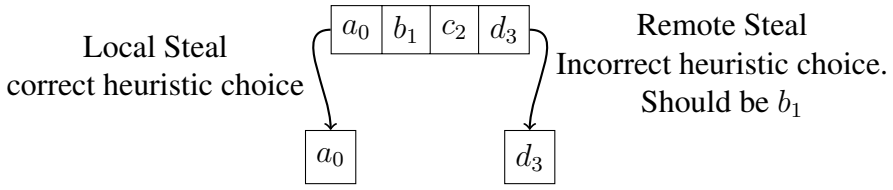


Figure 4.6: Deque-based scheduling breaks heuristic ordering. Subscripts represent discrepancies, lower is better.

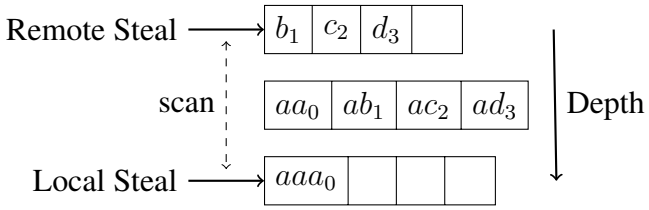


Figure 4.7: Depth-pool structure. Subscripts represent discrepancies, lower is better.

deque feature a private (lock-free) section and separate public section to further improve performance [35].

Unfortunately, deque work-stealing does not respect search heuristics (Section 2.1.4.1). To see why, consider the scenario in Figure 4.6 where each node is label with the number of discrepancies (at a particular depth) to show the heuristic search order. We observe that not only do remote steals not follow the heuristic ordering but they choose the heuristically worst possible node.

In practice deque work-stealing for tree search still performs well. In particular for cases where the search heuristics are not strong [117]. This is due to, by shuffling the order of tasks, diversity being added to the search. Increased diversity improves the chances we get lucky and find a solution early. This resembles discrepancy based search where heuristics are purposefully broken.

We argue that a more principled approach to work-stealing is necessary that maintains the heuristic order as much as possible (unless a user explicitly suggests otherwise). We require a structure that still allows the distinction between local and remote steals, while respecting the search heuristic ordering. To this end, we propose an alternative workpool, the *depth-pool*, as shown in Figure 4.7.

Here the deque structure is replaced by an list of first-in-first-out (FIFO) queues, one for each depth we have spawned at. By using FIFO queues the heuristic ordering is maintained at each depth. The depth-pool embodies the parallelism principle of stealing "high and left" (for remote threads), while allowing for a local help first approach.

The depth-pool does not **guarantee** the heuristic search order is fully maintained as two or more local threads can intersperse spawns at the same depth (e.g. $\{aa_0, ab_1, ba_0, ac_2 \dots\}$. It does however ensure that we do not bias towards heuristically poor choices; as deque

scheduling does.

Depth-pool is similar to the original Cilk workpool structure [34] that stored tasks based on their spawn "level" (i.e root = level 0, children of root = level 1 and so on) and stole at the "shallowest" possible depth while working from the "deepest". Our depth parameter is the depth in the search tree rather than a count of the number of spawns above, i.e. depth-pool is specialised for tree search. The Cilk version still removed from the tail of the queue (at each level) causing the same ordering issues present in a deque.

The depth-pool has two main disadvantages. All task spawn sites must now also be aware of their depth in the tree. This issue is minor as all spawns happen within a search coordination, hiding this detail from the user. The second disadvantage is the increased cost in managing the depth-pool, e.g. performing scans for work is $\mathcal{O}(n)$ and FIFO queues do not allow simultaneous inserts and steals from multiple workers. In contrast, many high performance, lock-free, deque implementations exists. We show in Section 6.5.1 that these performance issues do not occur in practice.

The requirement to have specialised workpools further motives the need for specialised tree search frameworks as opposed to relying on existing task-parallelism systems.

While the depth-pool has been designed specifically with search in mind, we do not see significant improvements over deque scheduling in practice (Section 6.5.1). It is currently unclear why this is the case.

4.5 Search Types

The search coordinations determine both how to perform depth-first tree traversal and how parallel tasks are generated and scheduled. That is, they encapsulate *Traversal* and *Spawn* rules of MT^3 (Section 3.3). As discussed in Section 4.1, useful work is only performed when the skeletons are given a specific search type, e.g. enumeration, decision, or optimisation (Section 2.1.3) and branch and bound variants.

In this section we discuss how the search types determine the node processing functionality for the search coordinations and the return type of the skeletons. The search types capture both the *Node Processing* and *Prune* rules of MT^3.

4.5.1 Enumeration Search Type

Enumeration searches accumulate information about the search space by visiting each node. They are captured by the *enumerate* rule of MT^3 (Section 3.3.6.1) that maps each node into a monoid and combines this with a globally stored monoidal value. The monoid type allows

Listing 4.5: Pseudocode for Enumeration Search Type.

```
global map nodeCounts

function enumeratingSearch(Coordination searchCoordinationEnumeration, Generator gen,
                           SearchSpace space, Node root, args...):
  // Calls a given search coordination with processNodeEnumeration added
  searchCoordinationEnumeration(gen, space, root, args...)
  return nodeCounts

function processNodeEnumeration(Node n, SearchSpace space, int currentDepth):
  nodeCounts[currentDepth + 1] ← nodeCounts[currentDepth + 1] + 1
  return NoPrune // We never prune in enumeration
```

many different styles of enumeration, e.g. creating a representation for all nodes, counting all nodes, counting left nodes etc.

The skeletons currently only support enumeration problems that want to count the number of nodes at each depth in the tree and this specialisation is described here.[9] This allows us to determine, for example, how many cliques of size k are present, without creating a representation for each clique. Only counting the number of nodes is not unusual as, given the huge size of search trees, generating and storing representations of the entire tree efficiently is difficult. For example, Fromentin and Hivert [118] suggest that storing all representations of numerical semigroups (Section 5.2.1.2) at depth 54 requires "several terabytes".

Using Haskell type notation, the type of an enumeration search is:

```
search enum :: Generator → Space → Node → Map Int Int
```

Where `Generator` is a Lazy Node Generator (Section 3.4) describing a specific search, the `Space` and `Node` parameters represent the global search space and root node, and the result is a map between depth and node counts.

Supporting enumeration searches requires 1) a global map structure to maintain counts of the nodes expanded at each depth, i.e. σ_e, and 2) ensuring the current depth is tracked at every node. Pseudocode for the search and node processing functions for enumeration are in Listing 4.5. For correctness the global `nodeCounts` map must be updated atomically to ensure thread saftey. In practice, to avoid the global map becoming a bottleneck, each worker manages a local map that is combined with the global map on task completion.

4.5.2 Decision Search Type

Decision problems search for a target node with a particular property in the tree. They are captured by the *decide* rule of MT^3 (Section 3.3.6.1) that, for each node, calls a `match` function to determine if it is the target. The skeletons currently assume that the `match`

[9] The skeletons can be extended to support the monoidal style of enumeration.

function is equivalent to an equality check on the objective value of a node rather than being passed explicitly by the user. For example, find a TSP tour of length (objective value) 100.

Using Haskell type notation, the type of the decision searches are:

```
search decision :: Generator → Space → Node → Objective → Maybe Node
```

As for enumeration, `Generator` parameter represents a Lazy Node Generator for a specific search and the `Space` and `Node` parameters represent the global search space and root node. The `Objective` parameter is initialised to the objective value of the target node. The result as an optional type that either returns a solution on a satisfiable instance or nothing for an unsatisfiable instances[10].

Two additional features are required for decision problems:

1. Early termination that causes all workers to stop if a solution is found. This is not a strict requirement and decision problems can be solved, inefficiently, by exploring the space fully even if a proof of satisfiability exists.

2. A method to obtain an objective value for a node, e.g a function $obj : X^* \rightarrow$ `Objective`, often implemented as a class member function for the node type. This value is compared with the user provided target objective for each node in the search tree. This requires the `Objective` type to allow equality checks.

Given a suitable *bounding function*, `upperBnd`, the decision skeleton may perform tree pruning based on the *target* objective value. This implements the p_d function of *prune-decide* in MT^3 and changes the search from backtracking to branch and bound.

Pseudocode for the search and node processing functions is in Listing 4.6. We assume a `terminateSearch` function (line 14) is available to perform a (distributed) termination proceedure.

4.5.3 Optimisation Search Type

Optimisation problems search for the *best* node (based on an objective function) in the tree. They are captured by the *optimise* rule of MT^3 (Section 3.3.6.1) that, for each node, calls an `improves` function to determine if the current node is an improvement over the best solution (so far).

Using Haskell type notation, the type of the optimisation instance is as follow:

```
search optimisation :: Generator → Space → Node → Node
```

[10] In the implementation of YewPar (Section 5.1.2) we report unsatisfiable by returning the root node.

Listing 4.6: Pseudocode for Decision Search Type.

```
global Node solution = Nothing

function decisionSearch(Coordination searchCoordinationDecision, Generator gen,
                        SearchSpace space, Node root, args...):
  // Calls a given search coordination with processNodeDecision added
  searchCoordinationDecision(gen, space, root, args...)
  return solution

function processNodeDecision(Node node, SearchSpace space, Objective requiredObjective,
                             function upperBnd):
  if node.getObjective() == requiredObjective then
    solution ← node
    // Early termination point for all workers
    terminateSearch()

  // If a bounding fn is provided
  if upperBnd(space, node) < requiredObjective then
    return Prune

  return NoPrune
```

As before, the `Generator` is a Lazy Node Generator for the application and the `Space` and `Node` paramters represent the global search space and root node. There is always a result for optimisation even if it the root node.

As with decision searches, we assume the objective can be obtained from a particular node, e.g. via a class member function. An additional parameter (not shown here) may be passed that determines if the search is maximising or minimising the objective value.

The main change required for optimisation is to track the current incumbent (best solution so far) and use this when determining if the current node is an improvement. As with the decision changes, a *bounding function* may be provided to enable branch and bound search. This implements the p_o function of *prune-optimise* in MT^3. For correctness, the incumbent read and update should be performed atomically to ensure monotonically increasing objective values that may occur due to race conditions.

There is no early termination for optimisation searches as we need to *prove* that no better value exists via exhaustive search. Tracking the incumbent can be done efficiently in practice as is shown in Section 6.4.

Pseudocode for the search and node processing functions is in Listing 4.7.

4.6 Summary

This chapter introduces a set of general-purpose skeletons for search. The skeletons encompass all three types of search (enumeration, decision, and optimisation) as well as a range of parallel search coordinations.

Listing 4.7: Pseudocode for Optimisation Search Type. Assuming maximisation problem.

```
global Node incumbent

function optimisationSearch(Coordination searchCoordinationOptimisation, Generator gen,
                            SearchSpace space, Node root, args...):
  incumbent ← root
  // Calls a given search coordination with processNodeOptimisation added
  searchCoordinationOptimisation(gen, space, root, args...)
  return incumbent

function processNodeOptimisation(Node node, SearchSpace space, function upperBnd):
  if node.getObjective() > incumbent.getObjective() then
    incumbent ← node

  // If a bounding fn is provided
  if upperBnd(space, node) <= incumbent.getObjective() then
    return Prune

  return NoPrune
```

The skeletons are designed with an abstract task-parallel framework (TSF) in mind (Section 4.2). By designing to TSF, the skeletons are not limited to a particular programming language or framework improving performance portability. The features required by such a framework may be derived from the formal model of tree search introduced in Chapter 3, as well as extensions to support scalability by distributed-memory support.

We show (Section 4.1) that search skeletons require a search type, e.g. enumeration, decision or optimisation, to give them meaning, and that the concrete behaviour of the skeletons can be specified as a particular search coordination. Four search coordinations are introduced that differ on their work generation and distribution methods:

Sequential (Section 4.3.3): Generates a single depth-first search task.

Depth-Bounded (Section 4.3.4): Converts any node above user specified depth d_{cutoff} to a task.

Stack-Stealing (Section 4.3.5): Allows workers to request work directly from other workers.

Budget (Section 4.3.6) Generates work after completing a user specified *budget* number of backtracks.

Search coordinations essentially form the *Traversal* and *Spawn* rules of MT^3 and the work generation features of the search coordinations may be succinctly captured as MT^3 spawning rules (Sections 4.3.4.1, 4.3.5.1 and 4.3.6.1), although full work distribution details, e.g. work-stealing, cannot.

Distributed work-stealing requires careful choice of workpool structures. In particular, the standard deque based work-stealing may go against heuristic search ordering. We show how

this occurs and introduce a new workpool structure, the depth-pool, that avoids breaking heuristic orders as much as possible in Section 4.4.

Changes introduced by selecting different search type parameters are shown in Section 4.5. In particular, search type parameters introduces global data transfers to manage state/solutions, e.g solution maps or global incumbents, and add functionality such as bounding or solution testing steps. That is, they encompass the *Node Processing* and *Pruning* rules of MT^3.

An implementation of the skeletons is given in Chapter 5 and analysed in Chapter 6. Chapter 7 shows the design of an additional, specialised, skeleton that provides replicable performance for branch and bound searches.

Chapter 5

Skeleton Implementation and Case Studies

This chapter introduces YewPar[1], a C++ framework realising the parallel skeletons of Chapter 4 for distributed-memory architectures, as well as a set of case study applications.

The features and implementation of YewPar are discussed in Section 5.1. Search type specific functionality and case study applications are introduced in Sections 5.2.1, 5.2.2 and 5.2.3. We show the skeletons are general by applying them to seven different case study applications: Unbalanced Tree Search (Section 5.2.1.1), Numerical Semigroups (Section 5.2.1.2), k-Clique (Section 5.2.2.1), Subgraph Isomorphism (Section 5.2.2.3), Maximum Clique (Section 5.2.3.1), Travelling Salesperson (Section 5.2.3.2) and Binary Knapsack (Section 5.2.3.3).

YewPar, and the case studies, are used in Chapter 6 to evaluate and compare the performance of the skeletons.

5.1 Implementation

We begin by describing a prototype search framework, HTSL, (Section 5.1.1) that shows TSF (Section 4.2) and the skeletons are general enough to be implemented in two different programming environments (Haskell and C++). The implementation of YewPar is then discussed in detail in Section 5.1.2 and Section 5.1.3.

5.1.1 A Prototype Haskell Framework for Tree Search

A subset of the skeletons: SequentialDecision, SequentialOptimisation, DepthBoundedDecision, and DepthBoundedOptimisation, have previously been implemented by the author in

[1]Pronounced You-Par. A Play on words of the Yew tree and PARallelism.

a prototype Haskell framework; unnamed in [4] and *HTSL* in [5]. Support for the Ordered skeletons of Chapter 7 is also available.

The Haskell skeletons are built on the HdpH framework [119], a distributed-memory asynchronous task-parallel Haskell. HdpH provides the low-level implementation of TSF (Section 4.2).

In practice TSF was derived from this implementation and this has heavily influenced the skeleton designs and the implementation of YewPar. For example, the laziness of the Lazy Node Generators initially came from exploiting Haskell's built-in laziness features, yet has proven to be more widely applicable.

The Haskell skeletons feature a less general programming interface of the form (adapted for readability):

```
1 -- Types
2 type Node = (Solution , Bound , Candidates)
3 NodeGenerator :: Space → Node → [Node]
4 BoundFn :: Space → Node → Bound
5
6 -- A search skeleton
7 Sequential.search :: Space → Node → NodeGenerator → BoundFn → Solution
```

Nodes (line 2) are always formed of three parts:

1. A `Solution` that encodes a branch in the tree, e.g. the current tour in TSP.
2. A `Bound` that encodes the bound for the node (typically the objective value), e.g. current tour distance in TSP.
3. `Candidates` that encode where the search can proceed, e.g. unchosen cities in TSP.

HTSL is less general than the skeleton presented in this work. There is no support for enumeration searches and it is assumed the searches are always branch and bound (i.e. `BoundFn` in line 7). The interface also assumes a finite representation for a candidate set, precluding infinite, but depth-limited, search spaces (e.g. the numerical semigroups case study of Section 5.2.1.2).

Although the Haskell skeletons provide a useful prototype, numerous reasons prompt the move to YewPar:

- HdpH provides limited control over low-level implementation details, such as schedulers and workpools, without changes to the HdpH library. This makes it difficult to add additional skeletons and to implement low-level features, for example, new workpool structures (Section 4.4).

- C++ generally provides higher baseline performance than Haskell (quantified to be around 2–6× slower in [4]) allowing larger instances to be tackled.
- High performance architectures often have limited support for Haskell applications and are designed primarily with C++ and Fortran tooling in mind.

The ability to implement (a subset of) the skeletons in two different programming languages and runtime environments gives confidence in their generality. The full set of skeletons could be implemented in HdpH given small changes to the programming interface to support enumeration searches and larger changes to the HdpH runtime to support a wider range of scheduling policies and workpools.

5.1.2 YewPar: A Framework for Distributed-Memory Tree Search

YewPar is a C++ parallel search framework that realises the parallel algorithmic skeletons of Chapter 4 and provides low-level components such as schedulers and workpools with which new skeletons can be created.

The standard C++ runtime, on its own, does not provide the features of TSF (Figure 4.2) due to a lack of support for distributed-memory parallelism. To achieve the required functionality, YewPar is based on HPX [28], a C++ standards compliant task-parallelism library targeting both shared and distributed-memory architectures.

5.1.2.1 HPX

HPX[2] is a library/runtime for shared and distributed-memory task-parallelism in C++. It is standards compliant, supporting both the C++11 and C++14 parallelism standards, and extends these to support remote operations and distributed-memory parallelism. By adopting a library approach, HPX does not require custom language/compiler toolchains and task-parallel constructs are exposed to an application developer making it ideal to base new frameworks on.

HPX is designed with Exascale HPC systems (10^{18} FLOPS) in mind and focuses on achieving scalability by exploiting asynchronous task-parallelism as much as possible. It has successfully been used in a wide variety of application areas including storm forecasting and astrophysics simulation [120].

The HPX programming model is based around creating many lightweight user-space threads (HPX-threads) and utilising a lightweight scheduler to multiplex these onto physical operating

[2]Not to be confused with the similarly named HPX-5, a C library based around the same ParalleX model as HPX.

systems threads. Not only are HPX-threads able to run locally, but they can be scheduled remotely via active messages [121]. That is, one locality can ask another to create a local HPX-thread to do some work.

All tasks in HPX are asynchronous and use futures [122] to represent results that may not yet be computed. Local synchronisation is performed by waiting on (groups of) future objects.

As controlling search order plays a large role in parallel tree search (Section 2.3.2.1), YewPar does not fully utilise the lightweight task scheduler. Instead we focus on controlling a few (large) search tasks (still represented internally as HPX-threads). Lightweight task and synchronisation features are still used for termination and knowledge exchange.

HPX features an adaptive global address space (AGAS) that assigns globally unique identifiers to HPX **components**. A HPX component lifts standard C++ objects into globally addressable objects (given properties around serialisation are met). Interaction with remote components is done via active messages that create a new HPX-thread (to call the member function) on the locality where the component is hosted. Resolution of physical addresses are handled transparently by the AGAS. The AGAS resembles partitioned global address spaces found in languages like X10 [111] or Chapel [112], and YewPar uses it as such[3].

A key limitation of HPX is the lack of built-in *distributed* load-balancing (apart from active component migration). Although work-stealing is used to load balance HPX-threads within a locality, no built-in support for inter-locality steals exists. This is not an issue as search problems require custom work-stealing scheduling functionality, e.g. to maintain heuristic orders (Section 4.4). We therefore implement distributed work-stealing scheduling directly in YewPar, i.e. at application-level. Implementing work-stealing at application-level has proved highly beneficial for experimentation and creating custom scheduling/workpool structures without requiring any HPX library/runtime modifications.

While we have chosen HPX, due to its ability to integrate with existing C++ tooling and search applications without requiring custom languages/compilers, implementing YewPar ontop of other languages is possible. Other task-parallel frameworks with similar features are X10 [111] and Chapel [112], that require custom toolchains (and patches to ensure work-stealing does not break heuristics, e.g. Section 4.4), or HabaneroUPC++ [123] that provides a similar interface as X10 without requiring a custom toolchain.

5.1.3 YewPar Features

YewPar provides four key features:

[3]The key difference is that AGAS supports (transparent) component migration at runtime, wheras PGAS implementations typically do not.

Listing 5.1: Worker thread scheduling loop.

```
1  global atomic running = true
2  global policy
3
4  function scheduler():
5    backoff = 0
6    forever:
7      if not running then
8        return
9      task = policy.getWork()
10     if task then
11       backoff = 0;
12       task()
13     else
14       backoff.increment()
15       suspend(backoff)
```

1. Search specific distributed work-stealing schedulers and workpools (Section 5.1.3.1).
2. Knowledge management: for example, storing and broadcasting global incumbents (Section 5.1.3.2).
3. The Lazy Node Generator interface (Section 5.1.3.4)
4. A library of skeletons (Section 5.1.3.5).

For efficiency and type safety, a large portion of YewPar is provided as header only files that are compiled to specialised implementations based on the specific types of a user's application (e.g Node type). By specialising at compile time we allow the optimiser to make additional type-based optimisations for improved performance. This template metaprogramming approach to skeletons resembles that of Quaff [104]. A disadvantage of the metaprogramming approach is increased compile times as a new skeleton is created whenever a template parameter is changed, including if the user passes a new node generator. In practice we have measured a compile time of around 1 minute to compile a single, fully optimised, skeleton for the Maximum Clique problem (Section 5.2.3.1). If the node generator type does not change then sharing of underlying components, e.g. the typed local registries, reduces the compile time of using additional skeletons within a single application. The amount of implementation sharing could be increased to further reduce compile times.

5.1.3.1 Application-level Scheduling

YewPar divides operating system threads (usually one per physical core) into two types:

Worker threads (workers): Workers run the scheduling loop shown in Listing 5.1 until they are terminated (i.e. `running` is set to false). The creation of workers is left to the particular skeleton coordination. For example, Depth-Bounded spawns N worker

threads at the start of search, whereas Stack-Stealing delays this until after a work-pushing phase.

Free threads: Any non-worker threads are free threads and are managed entirely by HPX. YewPar aims to keep at least one free thread available per-locality to allow timely processing of active messages, synchronisation, and global address space updates.

Within the worker scheduling loop idle workers request new tasks from a specific scheduling *policy*. The policy used depends on the type of coordination, for example Depth-Bounded uses a policy that manages distributed workpools (although could use a global workpool policy), whereas Stack-Stealing requires a policy that manages stealing between workers. Three scheduling policies are currently available. Like the skeletons, these are designed to be extensible.

Distributed Workpool (Figure 5.1(a)): This policy creates one workpool per-locality. The workpool may use either a deque or a depth-pool (Section 4.4) structure to manage tasks.

When work is requested the policy first attempts to find a task in the local workpool. If the local workpool is empty, the policy attempts to *steal* a task from the workpool of another locality. Steal requests are not forwarded to other nodes. If a steal fails the thief performs a backoff routine and then attempts a new steal.

The steal victim is first chosen as the locality of the last successful steal. If this victim contains no work then the next steal the victim is chosen uniformly at random. This *last steal* optimisation is effective as tasks tend to cluster at particular localities (i.e. those that have just spawned many tasks at some depth). We do not use other work-stealing optimisations such as low watermarking, where steals occur before the local workpool is fully exhausted, as this can cause heuristically-good tasks to be stolen and buffered rather than executed.

The Distributed Workpool policy is used for the Depth-Bounded (Section 4.3.4) and Budget (Section 4.3.6) skeletons.

Priority Ordered (Figure 5.1(b)): This policy creates a single, globally accessible, priority ordered workpool. All spawns, regardless of the spawning locality, are sent to this workpool. Likewise, all work requests target this single workpool. Work is retrieved in priority order. If the workpool is empty, workers periodically retry (with backoff) until work is found or the search terminates.

The Priority Ordered policy is central to the Ordered skeletons (Chapter 7).

Search Worker Manager (Figure 5.1(c)) This policy creates a set of worker manager components, one per-locality. No workpools are required.

Workers register with their local manager when they begin a search tasks and unsubscribe on completion. When a worker requests a task, the local manager issues a steal request to a (running) local worker at random. The victim worker is signalled to (try to) return an unexplored sub-tree via a direct channel to the manager[4]. If no workers are running locally then the local manager attempts a steal from a distributed manager at random (biased by the location of the last successful steal). The distributed manager invokes the same local steal procedure as before, this time returning the node(s) over the network.

The managers may contain a work buffer allowing N sub-trees to be stolen at a time. On a successful steal, one task will be sent to the requesting worker and the others buffered for later use. This reduces worker interrupts by providing a fast path for a manager to return a buffered task instead of performing a steal. YewPar uses a simple FIFO based buffer.

The Search Worker Manager is used for Stack-Stealing skeletons (Section 4.3.5).

5.1.3.2 Knowledge Management

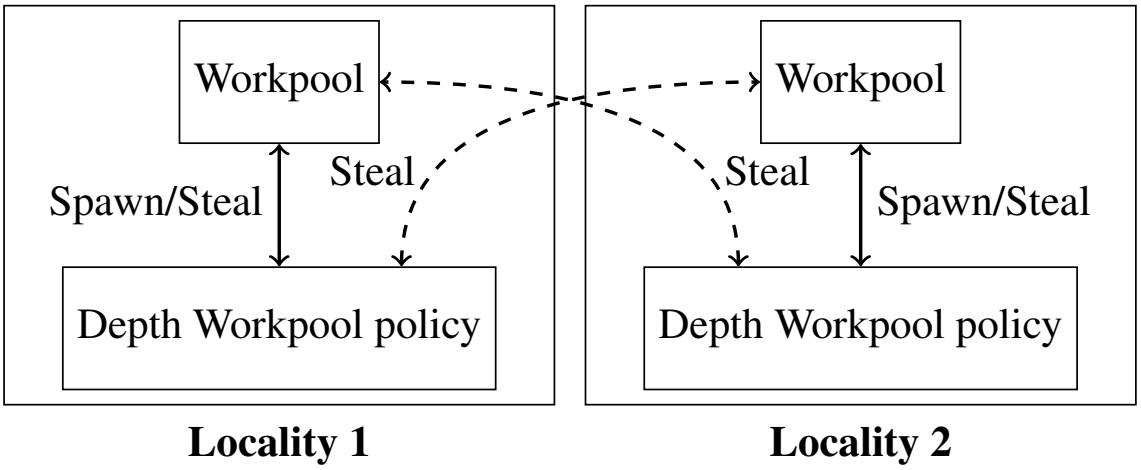
YewPar provides **each** locality with a (typed) global object that manages search specific variables that are shared between all workers of a locality. It maintains, among other things:

- The read only global search space, propagated to each locality at skeleton initialisation.
- Skeleton parameters such as spawn depths.
- Local node counts for enumeration problems.
- Local bounds for decision/optimisation search types.
- Termination flags.

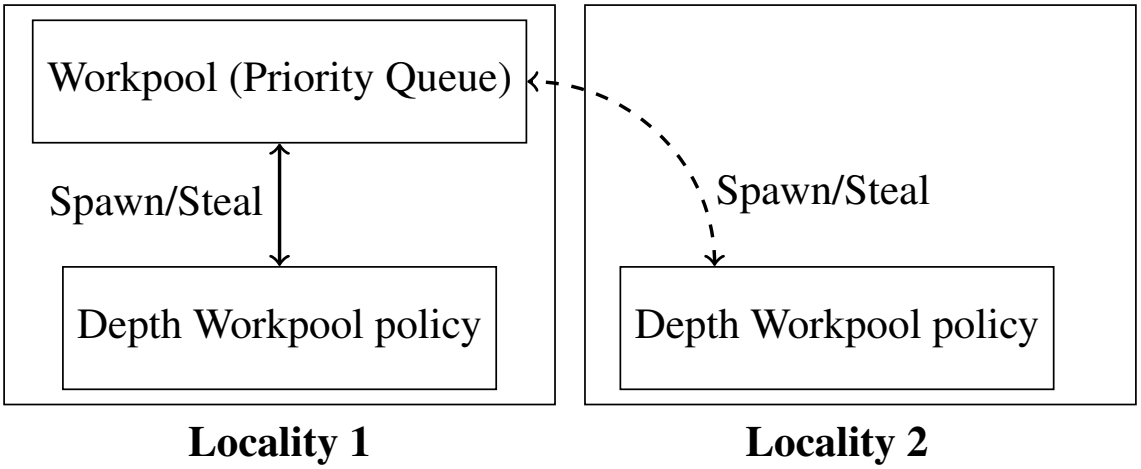
Although primary access to this state is local, it supports global updates via active messages. This is used, for example, when propagating new bounds or gathering the total node counts.

A global incumbent component keeps track of solutions for decision/optimisation problems. The implementation leverages the AGAS of HPX to build a globally accessible solution store. For decision search types the incumbent is updated once, if a solution is found. For

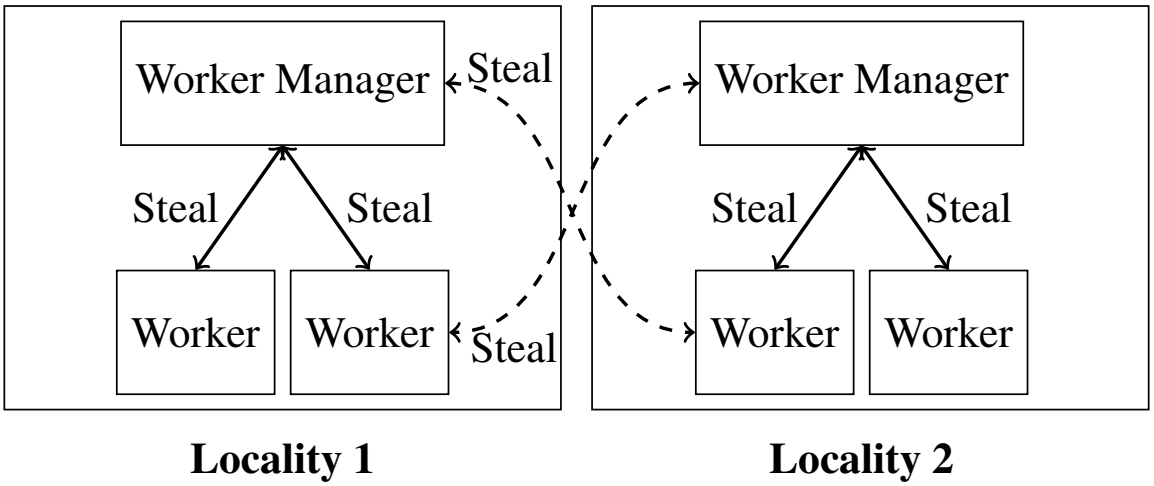
[4]An alternative approach is to have workers spawn into a local workpool structure on a steal and have the thieves steal from this structure. This approach matches the MT^3 spawn rule in Section 4.3.5.1.



(a) Distributed Workpool policy.



(b) Priority Ordered policy.



(c) Search Worker Manager policy.

Figure 5.1: YewPar scheduling policies.

optimisation problems the incumbent may be updated multiple times, and ensures only improved solutions are accepted. Currently, the incumbent only stores a single solution although it could be extended to support storing *all* solution nodes if required.

The store itself is hosted on the master locality, i.e. the one that initiates the search, although the exact location is transparent to the skeletons. We show in Section 6.4 that a global object is sufficient for storing the incumbent, due to infrequent irregular access patterns.

Access to the knowledge structures is limited to the skeletons and YewPar users never access this directly, ensuring all accesses are correctly synchronised.

5.1.3.3 Termination

The HPX future capabilities are used to manage termination in a tree–like manner. That is, child tasks explicitly notify their parent task on completion. Once the root task determines all its children are finished then all work must be complete and termination is safe. This form of termination is completely asynchronous.

Whenever a sub-tree is converted to a task the worker creating the sub-tree is responsible for allocating and managing a termination future. The *global* id of this future is captured by the spawned task and used to signal completion, including to remote localities.

Before a task terminates it checks if it is managing any unfulfilled termination futures. If this is the case, the worker creates a new HPX-thread to monitor the termination futures and send a signal to the parent when all are fulfilled. By creating a new HPX-thread the worker is free to continue with other search tasks. Free threads execute these lightweight termination tasks when they are ready and is a key reason why YewPar trys to maintain a free thread whenever possible.

5.1.3.4 Lazy Node Generators

Listing 5.2: Lazy Node Generator. Base Structure.

```
1  template <typename NodeType, typename Space>
2  struct NodeGenerator {
3
4    unsigned numChildren;
5
6    // When called, return the next child element
7    // Pre condition: numChildren < number of next calls
8    // number of calls is tracked by the search coordination, rather than the generator
         itself
9    virtual NodeType next() = 0;
10
11   // Derived types must have the following constructor signature
12   NodeGenerator(const Spacetype & space, const Nodetype & node);
13 };
```

User applications are constructed by passing a Lazy Node Generator (Section 3.4) to a search skeleton. All Node Generators are derived from the `NodeGenerator` class shown in Listing 5.2. The class is parameterised by both the type of a search tree node and the global, read-only, search space (line 1). Laziness is encoded by the `next` function (line 9), that should construct the next node when called. A generator can be made strict by computing all child nodes when the **generator** is constructed, storing them as a list in the class, and having `next` remove elements from the list. As generators are always constructed using the search space and a parent node they require a constructor with the signature of line 12.

Lazy Node Generators must always set their number of children (line 4) at construction. Internally the skeletons track the number of times the `next` method of a generator has been called and use this to determine when the generator is exhausted. An alternative approach would be to use an optional type to encode when a node is exhausted, as in the Haskell framework of Section 5.1.1 (that checks if the generators lazy list is `Empty` or `Cons`). This approach requires additional `next` calls when searching for work. For example, Stack-Stealing needs to call `next` for each generator starting from the bottom of the stack, until a non-null result is returned. By recording the number of next calls, the skeletons search for the first generator where number calls are less than `numChildren` without needing to ever call `next`.

The assumption that the number of children is known at generator construction limits the applicability of the implementation, but not the Lazy Node Generator interface. So far, we have not found an application where calculating the number of children at construction time is not possible or causes high overhead.

To see how this works in practice, we give an example Lazy Node Generator for the binary knapsack problem (described in Section 5.2.3.3) in Listing 5.3.

5.1.3.5 Skeleton API

YewPar implements the skeletons of Chapter 4 and Chapter 7. The skeletons are highly parameterised by types at compile time that allow specialised implementations to be compiled for each application. Typing information allows stack based memory allocation and compile time elimination of unused branches, e.g. if branch and bound is disabled then all pruning code is removed at compile time removing the conditional check at runtime. An alternative approach is to create type-agnostic skeletons, e.g. those that uses `void*`, however these provide less information to the compiler.

When calling a skeleton, the user first selects the coordination, e.g. Depth-Bounded, provides a Lazy Node Generator (Section 5.1.3.4), and chooses the type of search, e.g. decision. The skeletons are parameterised by additional settings that, for example, enable specific

Listing 5.3: Lazy Node Generator for Binary Knapsack.

```
struct KPSolution {
  vector<int> items;
  int profit;
  int weight;
};

struct KPSpace {
   vector<int> profits;
   vector<int> weights;
   int numItems;
   int capacity;
};

struct KPNode {
  KPSolution sol;
  vector<int> rem;
  int getObj() const { return sol.profit; }
};

struct GenNode : YewPar::NodeGenerator<KPNode, KPSpace> {
  int pos;
  const KPSpace & space;
  const KPNode  & parent;

  GenNode (const KPSpace & s, const KPNode & n) :
      pos(0), space(s), parent(n) {
    this->numChildren = parent.rem.size();
  }

  // Each node tracks the possible items that may be taken without
  // breaking the capacity constraint (the remaining items).
  // The next child is found by adding each of these possible items to the
  // knapsack in turn, creating a new solution (and set of remaining items).
  KPNode next() override {
    auto nextItem = parent.rem[pos];
    auto childSol = parent.sol;

    childSol.items.push_back(nextItem);
    childSol.profit += space.profits[nextItem];
    childSol.weight += space.weights[nextItem];

    ++pos;

    vector<int> childRem;
    copy_if(parent.rem.begin() + pos, parent.rem.end(), std::back_inserter(childRem),
              [&](const int i) {
                return newSol.weight + space.weights[i] <= space.capacity;
              });

    return { childSol, childRem };
  }
};
```

optimisations such as PruneLevel (Section 3.4) or allow switching between maximising or minimising optimisation search. These settings are built into the skeletons at compile time. Parameters, such as d_{cutoff}, are passed as runtime parameters (in a `Params` structure) making it easier for experimentation with these settings.

For decision and optimisation searches we place the additional requirement that a search node must have a `getObj` function that returns the current objective value of that node. This makes the objective function implicit in the types as opposed to passed as a skeleton parameter.

Listing 5.4 shows three different skeleton calls. `AppNodeGenerator` is an instance of a `NodeGenerator` (Section 5.1.3.4) and specifies a user application, e.g. Knapsack in Listing 5.3. `Node` is the node type for the application, i.e. the `NodeType` parameter of a Node Generator.

To improve usability, only the Node Generator has a specific template argument position (position one). All other parameters may be specified in any order and each has a default value. The return type of the skeletons is derived from the parameters given, for example, a `CountNodes` parameter returns a map of depths $\rightarrow$ nodecounts, while the same skeleton implementation called with an `Optimise` parameter will return the optimal search tree node (type `Node`).

Skeletons do not perform pruning unless a `BoundFunction` parameter is provided. The `BoundFunction` parameter takes a function type, where function types lift raw function pointers to the type level, allowing low-overhead function calls[5].

5.2 Case Study Applications

This section shows the reusable and general-purpose nature of the skeletons by presenting a set of seven case study applications featuring all types of search: enumeration (Section 5.2.1), decision (Section 5.2.2), and optimisation variants (Section 5.2.3).

5.2.1 Enumeration Case Studies

Enumeration problems fully explore a search space, accumulating information on nodes as the search progresses. As discussed in Section 4.5.1, the skeletons (currently) only support the case where the number of nodes at each depth (and no representations) are required.

YewPar implements enumeration searches as follows. At the start of each task, the worker allocates a map (implemented as a `std::vector`) to maintain the (local) depth to node

[5]Compared to polymorphic function objects such as `std::function` that are passed at runtime.

Listing 5.4: Skeleton API calling examples.

```
// Example 1: Branch and Bound Decision problem, using the Sequential search coordination.

// Runtime search parameters (using bound type int)
YewPar::Skeletons::API::Params<int> searchParameters;
// Decision problems need a target value to search for.
searchParameters.expectedObjective = x;

// Returns either "root" or a node with objective value "x"
Node sol = YewPar::Skeletons::Seq<
  AppNodeGenerator, // User application code
  Decision, // Decision search type
  BoundFunction<upperBound_func>, // Use Branch and bound with function upperBound_func
  PruneLevel // If a bound check fails don't explore children "to-the-right"
  >
  ::search(space, root, searchParameters);


// Example 2: Branch and Bound Optimisation problem, using the Stack-Stealing search
    coordination.

// Returns the "maximal" node.
Node sol = YewPar::Skeletons::StackStealing<
  AppNodeGenerator, // User application code
  Optimisation, // Optimisation search type
  PruneLevel, // If a bound check fails don't explore children "to-the-right"
  BoundFunction<upperBound_func> // Use Branch and bound with function upperBound_func
  >
  ::search(space, root);

// Example 3: Enumeration (Count nodes) problem, enumerating a tree till depth "x" only.

// Runtime search parameters (with no bound type)
YewPar::Skeletons::API::Params<> searchParameters;
searchParameters.depthLimit = x;
searchParameters.spawnDepth = y;

vector<uint64_t> counts = YewPar::Skeletons::DepthBounded<
  AppNodeGenerator, // User application code
  DepthLimited, // Only run to a fixed depth (set in searchParameters)
  CountNodes // Enumeration search type
  >
  ::search(Empty(), root, searchParameters);
```

count mapping. This structure is updated throughout the search and, just before finishing the task, the map is (atomically) combined with a shared per-locality map that is stored in the localities registry. Once the entire search is complete, the main thread gathers and combines the maps from each locality to return a final node count map.

Two enumeration case studies are considered: The Unbalanced Tree Search benchmark, and determining the number of numerical semigroups of genus g.

5.2.1.1 Unbalanced Tree Search

The Unbalanced Tree Search (UTS) benchmark [124] dynamically constructs synthetic irregular tree workloads. It is often used for evaluating new load-balancing techniques, e.g [125, 126]. Although there is significant work on UTS as an irregular parallelism benchmark, few of the new load-balancing and parallelism techniques have been adopted by the wider search communities.

Using a few parameters, UTS constructs search trees with different shapes, sizes, and imbalances, allowing for a potentially huge range of test instances. In the original version of the benchmark the total node count and the number of leaf nodes are reported. Here we only track the total number of nodes (as per Section 4.5.1).

Nodes in UTS are each represented by 20-byte descriptors that are used as random variables, alongside a tree type, to determine the number of children of a node. Descriptors for child nodes are generated using a cryptographic SHA-1 hash function [127] applied to the parent descriptor and the child index. This makes the process deterministic and reproducible, while also ensuring no method exists to calculate the size of the tree without searching it.

Different tree shapes are created by varying how the number of children are determined. Although the freely available UTS implementation [128] supports Binomial, Geometric, Hybrid and Balanced trees, we only evaluate using the Geometric tree type. This common in the existing literature and gives a wider range of branching factors. Geometric trees use a geometric distribution such that each node has n children with probability $p(1-p)^n$.

Keeping in line with the literature (e.g. [129, 130]), we use variants on the T1 sample tree instance included in the UTS repository. Although this does not express a wide range of tree types it allows progressively larger instances to be easily created. Because of resource constraints we choose trees that ensure a YewPar sequential runtime of less than two hours. In all cases we use the *FIXED* variant of the geometric tree type. This variant assumes child count equal to the branching factor and then adjusts based on a geometric probability function (that relies on the SHA-1 state of the parent node), such that there are between 0 and *branching factor* children for each node. The tree parameters are specified in Table 5.1.

To the best of our knowledge this is the first skeletonised version of the UTS benchmark.

Instance	Branching Factor	Depth	Random Seed
T1XL	4	15	29
T1XXL	4	15	19
T1XXL+	4	16	19

Table 5.1: UTS instances.

5.2.1.2 Numerical Semigroups

The second enumeration case study comes from computational group theory and asks the question "how many numerical semigroups are there of genus g"?

We use the following definition of a numerical semigroup from Bras-Amarós and Fernández-González [131]:

> "Let $\mathbb{N}_0$ be the set of non-negative integers. A numerical semigroup is a subset Λ of $\mathbb{N}_0$ which contains 0, is closed under addition and has finite complement, $\mathbb{N}_0 \setminus \Lambda$. The elements in $\mathbb{N}_0 \setminus \Lambda$ are the *gaps* of Λ, and the number $g = g(\Lambda)$ of gaps is the *genus* of Λ."

Intuitively, we can view a numerical semigroup as taking the non-negative integers and removing a finite number of (non-zero) elements, ensuring all remaining elements maintain closure under addition. For example $\Lambda = \{0, 2, 3, \dots\}, \mathbb{N}_0 \setminus \Lambda = \{1\}$, forms a numerical semigroup as $2{+}2 = 4, 2{+}3 = 5$ and so on. Whereas: $\Lambda = (\{0, 2, 3, 5, \dots\}, \mathbb{N}_0 \setminus \Lambda = \{1, 4\}$, is not a numerical semigroup as $2 + 2 = 4$ so the set is not closed under addition.

Numerical semigroups are often written in terms of their *generators*. A generator is described as the smallest set of integers required to create the group. For example, $\langle 1 \rangle$ generates $\{0, 1, 2, 3, \dots\}$ and $\langle 2, 3 \rangle$ generates $\{0, 2, 3, 4, \dots\}$. In all cases 0 is implicit.

Numerical semigroups arise in the study of the Frobenius (coin) problem that looks to determine the largest value that cannot be created using linear combinations of coins with specific denominations. For example, given coins of denominations 5 and 7 the largest quantity that cannot be created is 23, with $24 = 2 \times 7 + 2 \times 5$, $25 = 5 \times 5$ and so on. Additional uses of numerical semigroups are given by Assi and García-Sánchez [132] and include algebraic geometry and coding theory.

To find the number of numerical semigroups with a particular genus we build the tree of generators in Figure 5.2. At each expansion step, a *single* integer is removed from the current node's generator and one or more additional elements are added such that the new child generator forms a numerical semigroup. For example, if we remove 1 from the root Node Generator, we must add in both 2 and 3 (else 5 would be a gap). As each expansion consists

of removing a single element from the generator, i.e. creating a new *gap* in the numerical semigroup, the depth of a node corresponds directly to the number of *gaps*. That is, tree depth is equal to the *genus* of the numerical semigroup. To find the number of numerical semigroups of genus g we need to determine how many nodes there are at depth g.

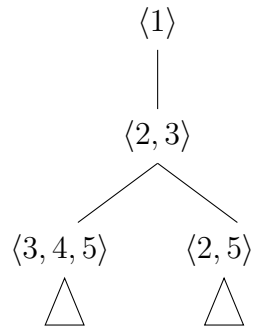


Figure 5.2: Beginning of the tree of numerical semigroups.

An interesting feature of this tree is its *infinite* size (as it is based on the infinite set of non-negative integers). This is in contrast to many search trees that are finite (yet huge). The skeletons support infinite trees by allowing search to be run to a fixed depth.

The implementation is based directly on the depth-first search approach of Fromentin and Hivert [118]. YewPar wraps the existing functionality [7] in a Lazy Node Generator to compute the child nodes. Underneath, the implementation uses many optimisations such as an efficient bitset encoding, bit parallel instructions, and loop unrolling for high performance.

The original implementation allows parallel searches using Cilk++ [26] limiting it to shared-memory[6]. Parallelism is introduced by spawning all children until a depth cutoff (the authors suggest $max\_depth - 11$ works well), in a similar manner to the Depth-Bounded skeleton. By integrating this existing implementation with YewPar, distributed-memory parallelism is added at a low engineering cost of a few additional node serialisation methods and some wrapper functions.

Unlike many of the other case studies, only one instance of this problem exists as the tree is the same in all cases. For experimentation we only consider a genus of up to $g = 50$ to keep sequential running times reasonable[7].

5.2.2 Decision Case Studies

Decision problems try to determine if a particular node, the target, is present in the search tree.

Decision problems allow early termination in the case of satisfiable instances. In YewPar this is managed by broadcasting a termination signal to all localities. This signal is stored in the local registry. The termination signal is explicitly checked by each worker at every search expansion step. If it is set then the worker performs any necessary clean up and exits the task without performing additional search. Due to the termination algorithm used by

[6] A Sagemath binding allows distributed parallelism using a map-reduce style of computation.

[7] $g = 49$ takes around 3900s sequentially. For $g = 50$ we only report scaling relative to 16 workers in (Section 6.9).

YewPar (Section 5.1.3.3), any tasks remaining when termination is signalled are still processed before the search completes fully. These remaining tasks do no work other than checking the termination flag on the first expansion step and sending task completed messages to parents (by filling futures)[8].

We only support the case where the target node can be determined by compared the objective value of the current node to the known target objective. If a solution is found, it is sent to a global incumbent object where it is read by the main thread at the end of the search.

We consider two decision problems: determining if a clique of size k is present in a graph (k-clique), to solve problems in finite geometry, and the subgraph isomorphism problem.

5.2.2.1 k-Clique

A *clique* in a (undirected) graph is a set of vertices where each vertex in the set is adjacent to every other vertex in the set. For example in Figure 5.3 the set $V = \{a, b, d, g\}$ forms a clique, but $V = \{a, d, c\}$ does not, as there is no edge from d to c.

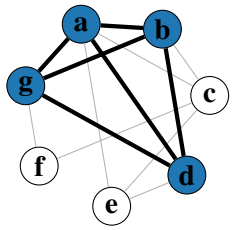


Figure 5.3: A graph, with its Maximum Clique $\{a, b, d, g\}$ shown.

In the k-Clique problem[9] we ask if *any* clique of size k exists in the graph. For example there is no 5-clique in Figure 5.3, but there is a 4-clique.

Clique search (enumeration, decision and optimisation variants) has applications in many domains including bio-informatics, chemo-informatics, planning and network analysis [133].

Our clique search implementation is a variant of San Segundo's BBMC [134]. We use the same efficient bitset encoding of adjacencies, but initially order the vertices in non-decreasing degree order, tie-breaking by vertex number, rather than random tie breaking (i.e. we use the MCSa1 algorithm from Prosser [10]). No recolouring is used (e.g. [135]).

The bitset encoding allows for instruction level parallelism through vectorisation of bitset adjacency calculations. This can be viewed as a form of parallel node processing (Section 2.3.1). Parallelism of this form is captured within the Lazy Node Generators and is orthogonal to the skeletons.

[8] An alternative approach would be to use the global address space and have a single *isSatisfied* future blocked on by a main thread, allowing further processing to happen before all tasks have completed. This alternative approach leaves the YewPar scheduler in an unknown state making it difficult to call multiple search skeletons in a single application run.

[9] The term k-clique is sometimes used to describe a clique with a weakened adjacency constraint that allows vertices in the clique if are at most k-edges from every other vertex in the clique. Here we always use k-clique to describe the decision variant of clique search.

5.2.2.2 Finite Geometry Instances

We apply k-clique search to a particular application: determining if a spread of the Hermitian variety $\mathrm{H}(4, q^2)$ exits. We consider the specific geometries $\mathrm{H}(4, 2^2)$ and $\mathrm{H}(4, 3^2)$ in this work.

We are not the first to apply parallel search methods to problems arising in finite geometry. The search for a finite projective plane of order 10 is a famous example [136] that was solved using a static approach where the search space was split ahead of time and ran on multiple machines.

Finite Geometry *Incidence geometry* is the study of structures that consist of a set of *elements* (points, lines, etc.) and an *incidence relation* between them. The incidence relation determines, for example, the points that belong to a line, the lines that intersect at a point, or lines that do not have points in common. *Finite geometry* considers incidence structures with a finite number of elements, and has applications including coding theory and cryptography [137].

Many finite geometries have associated graphs that represent vertices as elements, and adjacencies using the incidence relation. The study of (certain) substructures of a finite geometry can be reduced to studying the features, e.g. cliques, of the associated graph.

One finite geometry is $\mathrm{H}(4, q^2)$, which arises as the set of projective points of a non-degenerate Hermitian variety in four dimensions over the finite field of size q^2. An unknown result is if a *spread* exists in $\mathrm{H}(4, q^2)$ for all q, except $q = 2$ where it is known (computationally) that no spread exists.

A spread is a set of lines $\mathcal{L}$ such that every point is incident with exactly one element of $\mathcal{L}$. For $\mathrm{H}(4, q^2)$, a spread (if it exists) will have size $q^5 + 1$. Intuitively, a spread forms a partition of the points.

The question if spread exists in $\mathrm{H}(4, q^2)$ can be solved computationally (for a particular q) by constructing an associated a graph G where:

1. Lines of $\mathrm{H}(4, q^2)$ correspond to vertices in G.
2. Two vertices are adjacent in G if and only if the lines share no common point(s).

Using this mapping, a spread in $\mathrm{H}(4, q^2)$ corresponds to a clique of size $q^5 + 1$ in G. The k-clique implementation described in Section 5.2.2.1 can then be used solve this problem.

Two instances are considered, $\mathrm{H}(4, 2^2)$, where it is known that there is no spread[10], and the unknown $\mathrm{H}(4, 3^2)$. The graphs for these instances are constructed ahead of time using the

[10] A computational result, attributed to Andries Brouwer, appears to be unpublished. We have previously independently verified via computational search that there is no spread in $\mathrm{H}(4, 2^2)$ [5].

Geometry	Vertices	Edge Density	Automorphisms	Spread Size
$\mathrm{H}(4, 2^2)$	297	0.865	27,371,520	33
$\mathrm{H}(4, 3^2)$	6832	0.960	516,381,143,040	244

Table 5.2: Properties of the complement line graphs for $\mathrm{H}(4, q^2)$ where $q = 2, 3$

GAP computational algebra system [138] with the `FinInG` [139] and `grape` [140] packages. Key properties of the graphs are in Table 5.2, where edge density represents the ratio of edges to vertices ($D = \frac{2|E|}{|V|(|V|-1)}$).

Symmetry Breaking The number of automorphisms given in Table 5.2 represent the number of symmetries in the geometry. These symmetries are defined by the collineation group for $\mathrm{H}(4, q^2)$, that is well known. While a *brute force* approach can be used to check the existence of spreads in $\mathrm{H}(4, 2^2)$, this approach is unlikely to scale to larger geometries due to the large increase in the size of the search space.

The search space size can be reduced by accounting for symmetries. An important optimisation, used for algebraic applications as well as in domains like constraint programming [141], is **symmetry breaking**. Symmetry breaking reduces the size of the search space by ensuring symmetrical branches are visited at most once.

One method of symmetry breaking, performed in the branching step of a tree search, is orbital branching [142]. In orbital branching, child nodes are grouped into partitions based on the orbits of the automorphism group. At each step, only one node from each partition is branched on. This node is then *fixed* in further orbit calculations (effectively constructing a new automorphism group).

With Lazy Node Generators, this branching technique is transparent to the skeletons and may make use of systems such as GAP [138] to do the orbit calculations.

For the finite geometry case studies, we use orbital branching to break symmetries until a cut of depth $d_{symmetry\_cutoff}$ **before** running the parallel search. This process returns a set of new graphs to be explored. To show no spread exists, we must show that none of these new instances contains a clique of size $q^5 + 1 - d_{symmetry\_cutoff}$.

5.2.2.3 Subgraph Isomorphism Problem

Given two graphs, a *pattern* graph $\mathcal{P}$ and a *target* graph $\mathcal{T}$, the subgraph isomorphism problem (SIP) looks to determine if a copy of the pattern graph is present within the target graph. We consider the non-induced case for undirected graphs, where adjacent vertices in $\mathcal{P}$ are mapped to adjacent vertices in $\mathcal{T}$. An example of a (non-induced) graph isomorphism is

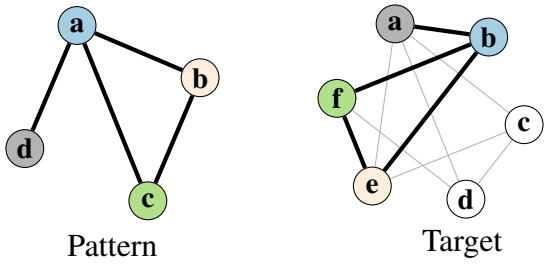


Figure 5.4: (non-induced) Subgraph Isomorphism problem example, a pattern and target graph with isomorphism $\{a \to b, d \to a, b \to e, c \to f\}$ shown.

given in Figure 5.4. Although there is in edge from a to e in the non-induced variant we are not required to include this in our subgraph. There may be more than one valid subgraph isomorphism for a $\langle$pattern, target$\rangle$ pair.

The subgraph isomorphism problem has important applications including chemo-informatics, bio-informatics, and pattern discovery in graph-databases.

Our implementation is based on a sequential algorithm from McCreesh and is an improved version of the algorithm presented by McCreesh and Prosser [143]. Each pattern vertex is initially assigned a domain of potential target vertices. At each step we select a pattern vertex (a *variable*) and try, in turn, to assign it all possible values in its domain. Each time an assignment is made the effects are propagated to all other domains to, e.g, avoid assigning the same vertex twice and ensure adjacency constraints. If the propagation fails, i.e. the assignment is inconsistent, then the next value in the domain is tried until a valid assignment is found. Backtracking occurs when no more valid assignments are possible. If a valid assignment is made for each pattern vertex (the objective in this case) then the problem is satisfiable and early termination can occur.

Pattern vertices are selected using the smallest domain first, tie-breaking on highest degree. Values are chosen in increasing degree order.

Instances come from those curated by Christine Solnon [144] and represent a range of different graph types. The instances used are detailed in Table 5.3, where the type represents the directory the instances come from. Instances were chosen at random from those that had a YewPar sequential run time of around one hour; ensuring a mix of satisfiable and unsatisfiable instances.

5.2.3 Optimisation Case Studies

Optimisation problems look to find a search node(s) that minimises or maximises an objective function f.

[11] To save space we refer to this as meshes-pat3-tar401 in the evaluation.

Instance	Type	Subgraph Isomorphism?
si2_r01_m600.06	si	true
all-meshes-CVIU11-pattern3-target401[11]	meshes	false
g18-g106	LV	true
g34-g79	LV	false
g36-g107	LV	false

Table 5.3: SIP Instances from Solnon [144].

YewPar implements optimisations problems using the global address space of HPX to store the current incumbent (best solution so far). At each search step the objective value of the current node is compared to the incumbent to determine if the incumbent should be updated. If the current node improves the incumbent then it unseats in incumbent in the global address space (assuming no better solution has been found in the meantime). Unlike in decision problems, optimisation problems do not allow early termination.

Incumbent updates are infrequent (Section 6.4) and the cost of querying the incumbent each search step, especially if it is on a remote locality, is high. Instead, YewPar stores the current bound information for the incumbent in the local registry object of each locality. On an incumbent update the new bound (without the solution) is broadcast to each locality which updates the local bound if no better solution has been found.

We consider three optimisation problems: finding the maximum clique in a graph (Section 5.2.3.1), finding the lowest cost tour in the travelling salesperson problem (TSP) (Section 5.2.3.2), and finding an optimal packing in the binary knapsack problem (Section 5.2.3.3).

The implementation of the maximum clique problem is state of the art, while the TSP and binary knapsack implementations use simple algorithms that show the generality of the approach as opposed to aiming for the most efficient implementations.

5.2.3.1 Maximum Clique

The maximum clique problem is the optimisation variant of the k-clique problem described in Section 5.2.2.1. It asks for the largest k where a k-clique(s) is present in the graph.

The implementation uses the same state of the art algorithm as the k-clique implementation (Section 5.2.2.1). Changing the existing clique search algorithm from decision to optimisation requires a *single* change at the skeleton call, and no changes to the clique search itself, further highlighting the usefulness of the high-level approach. The following subset of the DIMACS clique challenge instances [145] are used for evaluation[12]:

[12] A larger set of DIMACS clique instances is used to evaluate the overheads of the Lazy Generator API in Section 6.3.

- brock400_1.clq
- brock400_2.clq
- brock400_3.clq
- brock400_4.clq
- brock800_1.clq
- brock800_2.clq
- brock800_3.clq
- brock800_4.clq
- MANN_a45.clq
- p_hat500-3.clq
- p_hat700-3.clq
- sanr400_0.7.clq

This subset is chosen to ensure a sequential runtime of between a few minutes and two hours.

5.2.3.2 Travelling Salesperson

The travelling salesperson problem (TSP) is a classic optimisation problem. Given a set of cities and the distance between each par of cities, we want to find the shortest tour where each city is visited only once and the salesperson returns to the starting city. We consider only *symmetric* instances where the distance between two cities is the same when travelling in both directions.

TSP has many important applications including vehicle routing and efficient drilling of electronic PCBs [146].

In practice, TSP is often solved using linear programming and cutting plane techniques (branch and cut). These techniques form part of the popular Concorde solver [147] that has been used successfully to solve instances with thousands of cities.

We use branch and bound here, where children of a node represent all cities that have not yet been added to the tour. Branching on a particular city adds it to the current tour. At each step a bound is calculated as the weight of the minimum spanning tree of the remaining cities added to the current tour length. The minimum spanning tree is calculated using a variant of Prim's algorithm [148], where only the weight is calculated, without storing the spanning tree itself. The bound is pre-initialised to the result of a greedy nearest neighbour search. We assume no heuristic ordering on the candidate cities, instead working in increasing order of city labels.

This is a proof of concept implementation, based on trivial branching and pruning functions. It shows the generality of the approach, but is far from state of the art. In particular, it does not remove symmetrical tours, e.g. 1234 and 1432, other than by fixing the starting city.

Problem instances are created at random using the DIMACS TSP challenge instance generator `portgen` [149]. The instances are named as `rand_<numcities>_<random-seed>.tsp`, and are:

- rand_35_37662.tsp
- rand_36_46956.tsp
- rand_37_14763.tsp
- rand_39_16423.tsp

They have a sequential runtime of between 30 minutes and one hour.

5.2.3.3 Binary Knapsack

Knapsack packing is another classic optimisation problem. Given a fixed size container and a set of items, each with a weight and value, we want to find the (sub-)set of items that should be added to the container such that the total value maximised.

Knapsack problems have important applications such as bin-packing and industrial decision making processes [150].

We consider the 0/1 (binary) knapsack problem, where an item is either added to the knapsack or left. Variants of the knapsack problem exist [151] that, for example, allow items to be chosen multiple times, fractional items to be selected, or multiple knapsacks to be filled.

In practice, knapsack problems are often solved using branch and bound, dynamic programming or core methods [152]. We use branch and bound, where the children of a node are all possible items that have not been previously considered and do not break the capacity constraint. Branching on an item represents adding it to the knapsack.

At each step a bound is calculated using a linear relaxation [153] where, instead of solving for $i \in \{0, 1\}$ (i.e. take i or leave i), we instead solve fractional knapsack problem where $i \in [0, 1]$. As the greedy fractional approach is optimal, it provides an upper bound on the maximum potential value. For efficient bounds calculation, and as a child ordering heuristic, profit density ordering is used such that $\frac{p_i}{w_i} \geq \frac{p_j}{p_j}$ if $i < j$, tie breaking first on maximum profit, then minimum weight, then item number. Due to the profit density ordering, if a bound check fails then all nodes "to-the-right" should also be pruned, i.e. the PruneLevel optimisation of Section 3.4 is used.

As with the TSP implementation, this is a proof of concept to show the generality of the approach and is not a state of the art approach.

Although the knapsack problem is $\mathcal{NP}$-hard, many knapsack instances are easily solved on modern hardware. Methods exist for generating *hard* instances, e.g. [154]. Here we use a subset of Pisinger's pre-generated hard instances [155] shown in Table 5.4. The instances are chosen to have a sequential runtime of less than one hour.

Instance Name	Type	Number of Items
knapPI_11_100_1000_28.kp	Uncorrelated span(2,10)	100
knapPI_13_200_1000_48.kp	Strongly correlated span(2,10)	200
knapPI_14_200_1000_69.kp	mstr(3R/10, 2R/10, 6)	200

Table 5.4: Knapsack instances from Pisinger [155]. Types correspond to the generation methods described in [154].

Chapter 6

Evaluation

This chapter evaluates the performance of the skeletons within YewPar (Section 5.1.2) on the case study applications described in Section 5.2. Evaluations are performed on a 17 locality Beowulf cluster described in Section 6.1.

We begin by discussing caveats when evaluating parallel search (Section 6.2), in particular how the non-fixed workloads of branch and bound applications make it difficult to reason about parallel speedups.

Section 6.3 compares the Sequential skeletons to a hand-written Maximum Clique implementation to show the overheads of moving to the generalised searches provided by the skeletons, which is shown to be low (around 6.1% slowdown on average).

One challenge of searches is the propagation of knowledge to all workers as it is found. Section 6.4 studies global knowledge management using the PGAS model with bounds broadcasting and shows this method is appropriate for knowledge exchange on medium sized clusters (255 workers), due to the limited number, and spread out nature, of updates.

The three skeletons, Depth-Bounded (Section 6.5), Stack-Stealing (Section 6.6) and Budget (Section 6.7) are studied in isolation to show how user provided tuning parameters impact performance for to 120 workers and a detailed study of work-stealing performance counters. A comparison of the skeletons follows in Section 6.8.

Finally, to show how the skeletons perform at scale, larger test instances such as the search for spreads in $\mathrm{H}(4, 2^2)$ are performed on 255 workers (17 localities) in Section 6.9.

6.1 Experimental Setup

All evaluations are performed on a Beowulf cluster consisting of 17 localities each featuring dual 8-core Intel Xeon E5-2640v2 CPUs (2Ghz), 64GB of RAM and running Ubuntu 14.04.3

LTS. Exclusive access to the machines is used and we ensure there is always at least one physical core per worker thread (hyper-threading is not used). Threads are assigned to cores using the default mechanisms of the HPX runtime[1] and Linux. All applications are compiled using gcc 7.3.0 which supports the required C++17 features of YewPar.

Complete source code for YewPar is freely available [105]. Nix [156] environment scripts (for reproducible environments), experiment scripts, and open access data underpinning the experiments are also available [157].

6.2 Caveats when Evaluating Parallel Search

Analysis of parallel search, particularly those that support pruning, is difficult. Many of the common techniques used to analyse parallel applications do not apply to parallel search due to speculative parallelism. Speculation causes the total work to be dependent on both the *order* tasks are executed as well as the number of workers, i.e. we can speculate more with more workers. For enumeration problems, and non branch and bound searches, the workload is fixed and does not suffer from these effects.

A non-fixed workload makes it difficult to apply Amdahl's law [158], as the sequential portion of an application is also not fixed. Likewise Gustafson's law [159], while dealing with applications with non-fixed workloads, cannot be applied due to the non-deterministic nature of applications in $\mathcal{NP}$, i.e. there is no way to pick an instance with x times the workload of another. With this in mind, although we show scaling results in the analyses to gain insight into how search performance improves with workers, we should not expect a w times speedup on w workers.

Elements of non-determinism e.g. the workpool they steal from and the tasks they spawn (depending on bounds), causes the task *order* to differ between runs. As this causes different *workloads* per run it can lead to large amounts of variability in runtime measurements.

This can cause difficulty for experimental analysis. One way of dealing with this variance is to average over many measurements to reduce the effect of non-determinism. In the analyses presented here we opt for averaging over a small number of samples (usually 5 unless stated) due to resource constraints. It is unlikely this small number of samples truly controls for this non-determinism[2].

The guiding principle behind the analyses is this. Given resource constraints, instead of presenting a large number of samples for a small number of instances (to truly control for

[1] Version 1.0, commit `51d3d0`.

[2] In Chapter 7 we including scaling runs for a higher number of samples (30), that shows similar average behaviour to the 5 sample runs in this section.

non-determinism), we instead leverage the skeletons to present a large number of different applications/instances with fewer samples.

In Chapter 7 we show a method for replicable parallel search runtimes, however this comes at the cost of performing less well in general.

6.3 The Cost of Generality

Before evaluating the parallel skeletons, we quantify the cost of moving from a hand written, application-specific, sequential implementation, to using the general-purpose Sequential skeleton. We explore this using the Maximum Clique benchmark (Section 5.2.3.1), as it is a) based on a state of the art algorithm, and b) derived as closely as possible from a sequential implementation by McCreesh [160]. In both cases we fix the bitset sizes to 32 words to handle all instances without requiring multiple compilations as is performed by the template system in McCreesh's original code. The implementations are evaluated using a 55 instance subset of the DIMACS clique benchmarks [145] that run under one hour using the Sequential skeleton.

Table 6.1 shows a subset of the results where the runtime is greater than one second. Results are reported as the mean of 10 runs. As the runs are fully sequential we get a strict search ordering and limited variance in the results.

These results show that there is a cost for generality but in the majority of cases it is small, almost always less than 10% of the non-skeleton sequential runtime, with a (geometric) mean slowdown of just over 6.1%.

Application profiling shows the majority of this overhead comes from dynamic memory allocation in the generator's `next` function. This highlights a limitation of the current implementation. Node Generators currently make no distinction whether or not `next` is called to continue traversing the tree by the same worker or to generate new work e.g. on a steal. Because of this a generator **must return a copy of a node** rather than allowing updating in place, as in many hand written implementations. Nodes that contain dynamically allocated structures require memory allocation (and copies) on each next call. While allocation is generally quick, given the large number nodes in search trees the overheads quickly multiply.

In the Maximum Clique implementation this manifests itself in the vector that tracks the members of the current clique. In the fully sequential version a single vector is allocated ahead of time, vertices are added on expansions and removed on backtracks, requiring no memory allocation. The YewPar version on the other hand copies the current member vector and adds the next choice on each expansion step. Likewise on backtracks memory must be freed.

Instance	Hand-Written C++	YewPar	Slowdown (%)
MANN_a45	200.04	200.35	0.16
brock200_1	1.3	1.49	12.61
brock400_1	675.67	737.47	8.38
brock400_2	493.68	539.74	8.53
brock400_3	391.07	428.86	8.81
brock400_4	188.36	205.46	8.32
brock800_4	2,694.15	2,900.21	7.11
p_hat1000-2	252.26	264.62	4.67
p_hat1500-1	3.18	3.42	7.02
p_hat300-3	2.83	3.05	7.42
p_hat500-3	263.6	277.03	4.85
p_hat700-2	5.08	5.3	4.14
p_hat700-3	2,606.12	2,704.74	3.65
san1000	2.54	2.46	−3.22
san200_0.9_2	1.16	1.25	7.64
san200_0.9_3	29.41	31.72	7.29
san400_0.7_2	4.31	4.6	6.33
san400_0.7_3	2.43	2.61	6.64
san400_0.9_1	57.83	59.22	2.36
sanr200_0.9	69.86	74.63	6.39
sanr400_0.7	178.07	198.23	10.17
Mean			6.1

Table 6.1: Maximum Clique: hand-written sequential vs. Sequential skeleton Runtimes (s).

The tree generator API has low overheads for applications with limited memory requirements at each node. For application domains with greater memory requirements, such as SAT solvers that often use structures such as in watched literal schemes [161], maintaining a copy at each node would be impractical; significantly degrading perfromance and increasing memory requirements. An improved Node Generator API would be possible that could signal to the user whether a copy needs to be returned, i.e. on a parallel steal, or if updating in place can be performed. Given that our case study applications all feature limited per node memory requirements, we do not explore this alternative API here.

6.4 Global Incumbent Management

A key feature of branch and bound optimisation searches is the ability to use knowledge gained in one area of the search tree to influence the rest of the tree, i.e. through pruning. In YewPar knowledge transfer is performed using a mix of the global address space *and* (asynchronous) broadcast messages. Given a goal of the skeletons is to be scalable, we might ask if performing knowledge updates in this manner is viable at scale?

In the following experiment we run a subset of Maximum Clique instances on 255 workers (17 localities) and track both the number of incumbent updates (corresponding also to bounds broadcasts) and the time at which the update message was received. 255 workers represent the worst-case number of incumbent updates, i.e. when we have the most opportunity for parallelism. Results are gathered using Depth-Bounded with $d_{cutoff} = 2$.

Figure 6.1 shows the mean number of incumbent updates[3] for each instance, split into those that successfully updated the global incumbent and those that failed. Failure to update the incumbent implies that another worker found a better solution before the update message arrived.

For many instances the total number of incumbent updates are small, often less than 50. There does not seem to be any correlation between the total application runtime and the number of updates, e.g. p_hat500-3 performs 148 updates in 1.1s whereas brock800_3 performs 29 in 15s. The large number of incumbent updates for the p_hat instances looks to be caused by their large Maximum Clique sizes (50 and 62 vertices), which places a upper bound the number of successful updates.

For instances with a higher number of incumbent updates we observe that a greater proportion of these are unsuccessful updates. This appears to be caused by many workers finding improved results early in the run, i.e. most branches are proving somewhat beneficial, leading to a large number of updates near the beginning of search.

[3]Rounded up to the nearest integer.

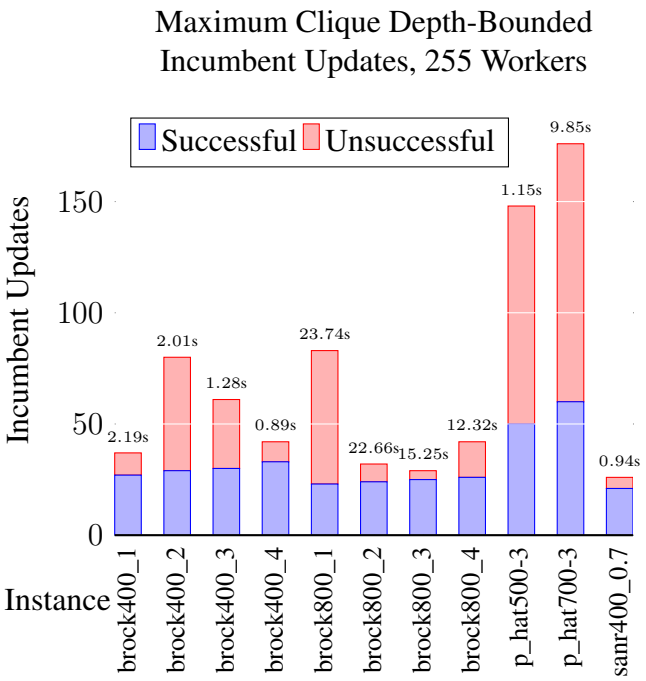


Figure 6.1: Maximum Clique: global incumbent updates using 255 Workers. Total runtime shown above the bar.

We see this clustering effect near the start of search more clearly in Figure 6.2 that plots the time[4] each update is processed by the global incumbent component. Many updates occur in the first couple of milliseconds in all searches. This is due to two implementation choices. Firstly, we update on *any* improved solution not just those at leaf nodes[5]. Secondly, bounds are not pre-initialised. For practical problems approximate methods approaches are commonly used to find a good initial bound before performing an exact search. With these in place we would expect to see a reduction in both the total number of updates and the clustering effect at the start of search.

In most cases unsuccessful updates cluster together around successful updates. This is as expected as an unsuccessful update implies that the local knowledge, of the unsuccessful worker, is currently out of date, i.e. due to a successful update occurring from a different worker and the broadcast message not yet arriving at the unsuccessful worker. brock800_1 appears to go against this, with three data points showing unsuccessful updates with no obvious successful updates occurring nearby. It is possible that a bound broadcast message has been delayed in reaching a particular node.

Given the low number of incumbent updates, and the fact many of these are due to implementation choices or lack of bounds initialisation (i.e. in practice it could be even lower), the

[4]Where time is measured from the creation of the incumbent object (roughly equal to the start of search).

[5]Changing to only update the incumbent/bounds on leaf nodes is a trivial implementation change. In practice, an implementation should allow toggling between both. For example, in TSP we know there is never a full tour on a non leaf node, with this setting we can avoid checking nodes we know will never cause an update.

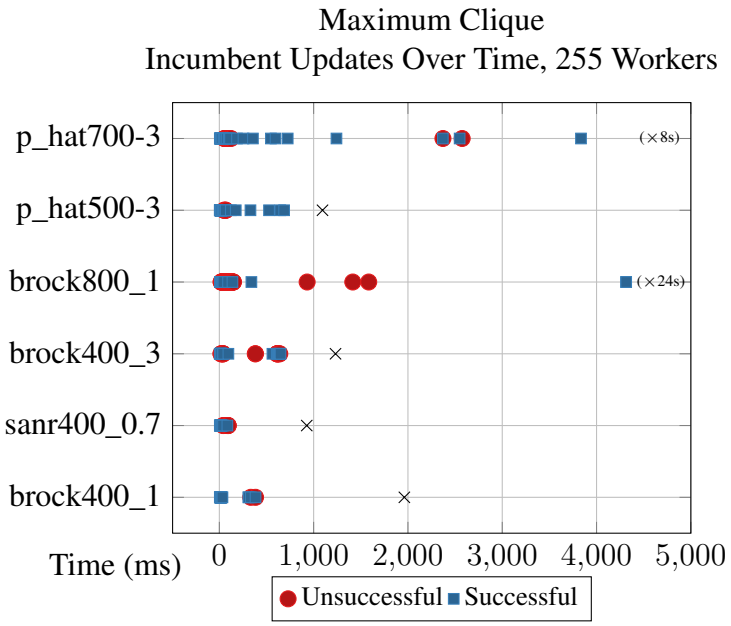


Figure 6.2: Maximum Clique: incumbent updates over time. × represents the final running time for the instance.

use of a global address space incumbent and global broadcast of bounds appears to be an appropriate method that likely scales to much larger setups. YewPar, using the PGAS and broadcasts, handles this small number of infrequent updates with ease and there does not appear to be any contention issues. Delays in messages, while detrimental to performance, do not affect the correctness of search and updates can be performed completely asynchronously without specifying rendezvous points.

Incumbent sizes in the case studies are small. For larger incumbent sizes this method may not be appropriate due to increased bandwidth requirements.

6.5 Depth-Bounded

The Depth-Bounded skeletons (Section 4.3.4) convert any node above a depth cutoff, d_{cutoff}, to a task. A major disadvantage of this search coordination is the requirement to choose a suitable d_{cutoff}.

For the one worker case, as depth increases we expect an increase in parallel overheads, i.e. the time to add and remove work to/from the workpool (in this case the depth-pool described in Section 4.4). Figure 6.3 shows this increase in one worker overheads when d_{cutoff} is varied between 0-8, where $d_{cutoff} = 0$ is equivalent to a sequential search. For Numerical Semigroups, we use depths of between 0-35 as the tree is known to be narrow near the root. Results are reported as the median over 5 runs with error bars reporting the minimum

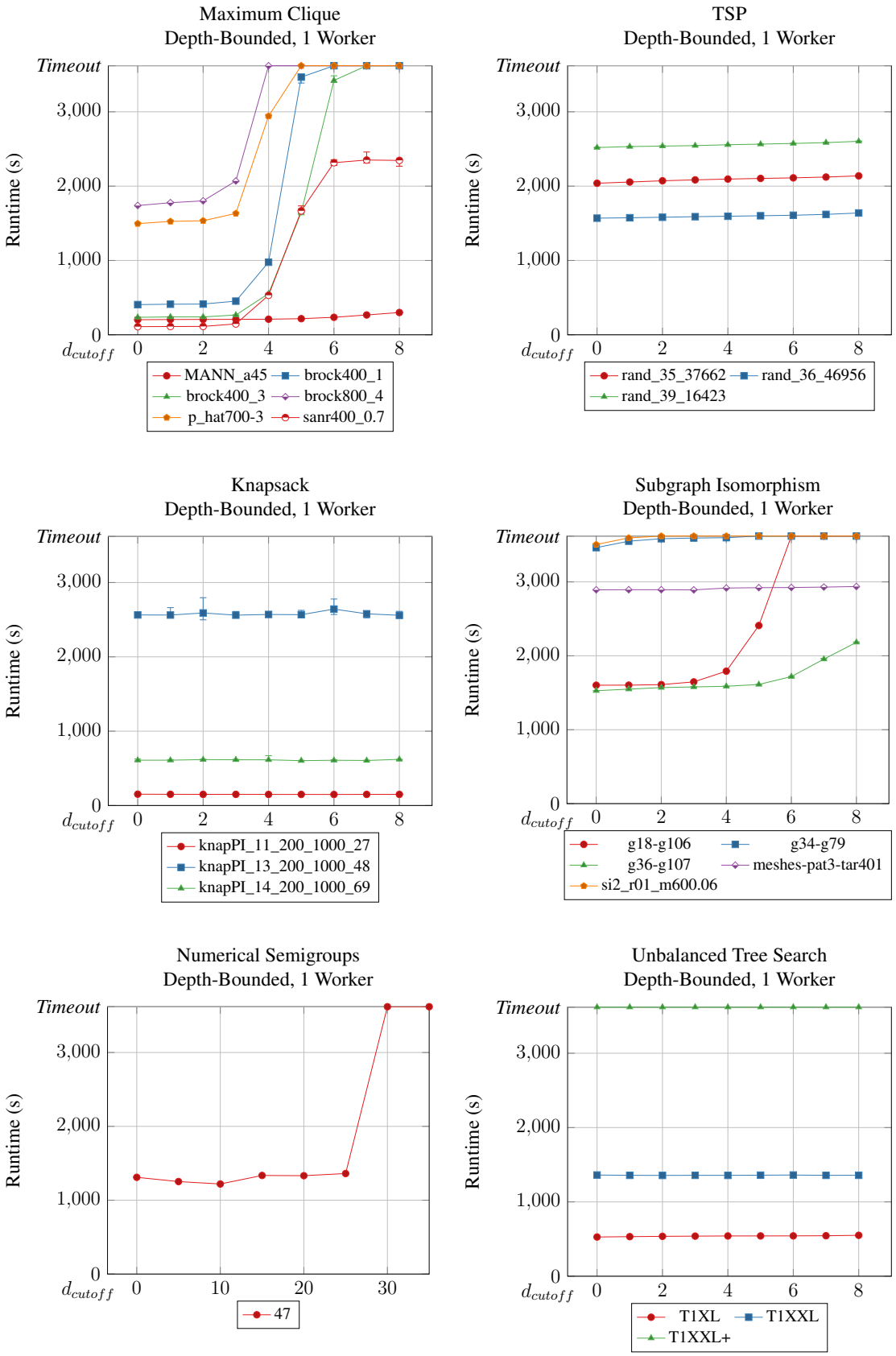


Figure 6.3: Impact of changing d_{cutoff} on the 1 worker runtimes for Depth-Bounded. Error bars show minimum and maximum runtime.

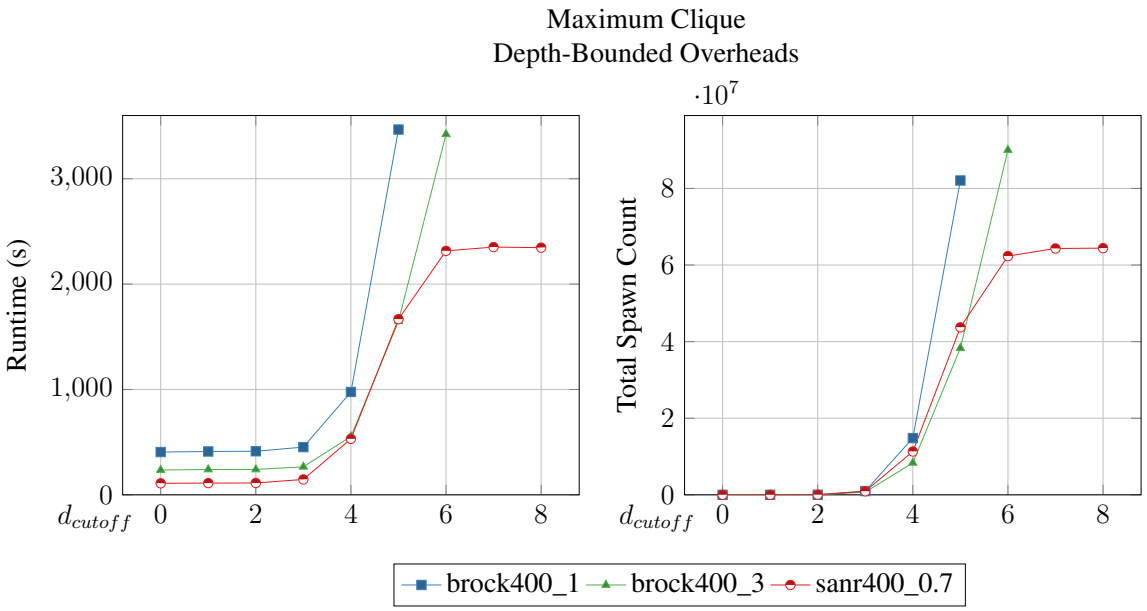


Figure 6.4: Maximum Clique: total task overheads when increasing d_{cutoff}.

and maximum runtime measured. As work is being removed from the (single) workpool in sequential order the benchmarks are not subject any search ordering effects.

The results are as expected, as d_{cutoff} increases we incur a runtime overhead cost to spawn/schedule the additional tasks. In many cases the overheads are small, for example Knapsack, TSP, UTS and much of SIP have small, linear, overhead increases. Maximum Clique shows many cases where, after some linear increases in runtime, the runtimes increase significantly e.g. brock400_1 suffers a slowdown of more than 2000s when moving from $d_{cutoff} = 4$ to $d_{cutoff} = 5$. As shown in Figure 6.4 this is caused by a likewise large increase in the number of tasks generated at these depths. The fact sanr400_0.7 stops increasing task counts at $d_{cutoff} = 6$ is likely due to additional pruning or significant narrowing of the branching factor of child nodes.

Numerical Semigroups shows interesting behaviour between $d_{cutoff} = 0$ and $d_{cutoff} = 15$ where the total runtime *decreases* even though there are more tasks to manage. It is not clear what causes this effect. One possibility is that the sub-trees fit better into memory, e.g. on a single page, but we have been unable to confirm this. As with Maximum Clique, due to rapidly increasing task counts Numerical Semigroups times out with a d_{cutoff} of more than 25.

Figure 6.5 shows a similar d_{cutoff} sweep, this time for the 120 worker case (15 workers over 8 localities). For a $d_{cutoff} = 0$ we get a sequential search plus the overheads of running additional schedulers (which will attempt to steal, but never be successful). The general trend of the results is as expected, as we increase d_{cutoff} we add more opportunity for parallelism until we reach a point where the overheads of managing additional tasks dominate the runtime.

Depth-Bounded performs particularly poorly on the Knapsack and Numerical Semigroups

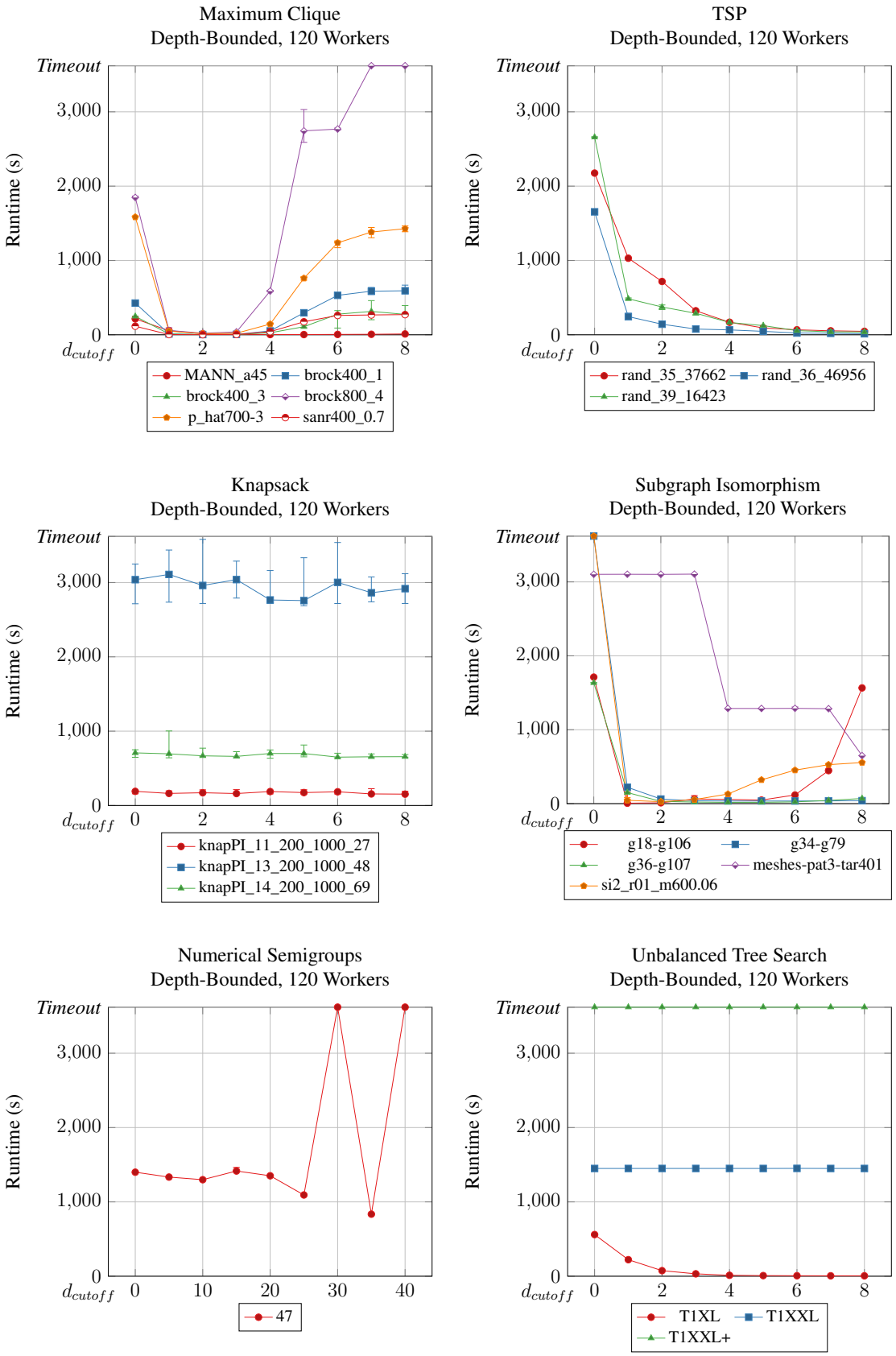


Figure 6.5: Impact of changing d_{cutoff} on the 120 worker runtimes for Depth-Bounded. Error bars show minimum and maximum runtime.

instances. For Knapsack this is likely due to setting d_{cutoff} too low, as in most cases the solution is somewhere deep in the leftmost branch (due to profit density ordering being a particularly strong heuristic). Numerical Semigroups similarly looks to have too low a setting of d_{cutoff}, with runtime improvements being observed with a d_{cutoff} of 25 and 35. This may be expected given to the low branching factors of the search tree at low depths (see Figure 5.2). Interestingly, a timeout is observed for $d_{cutoff} = 30$, even though runtimes are observed at d_{cutoff} of 25 and 35. It is currently unclear what causes this effect.

For many instances, particularly those in TSP, UTS, and (a subset of) SIP, the runtimes quickly approach a few seconds and continuing to increase d_{cutoff} has limited effect. In these cases, where the average branching factor tends to be small, it is often better to choose a slightly higher depth than is expected such that the initial poor scaling area is avoided. This is not the case for instances with large average branching factor, e.g. brock800_4, where over estimating spawn depth can easily lead to slowdowns.

It is not clear why the T1XXL UTS instance does not scale well. Unlike Knapsack that does not scale well due to the position of the solution, as UTS is an enumeration problem it is most likely due to $d_{cutoff} = 8$ being too small to introduce enough parallelism. This is backed up by the small increase in one worker overheads corresponding to limited tasks being spawned.

The improvement of the 120 worker parallel runs relative to the 1 worker case is shown in Table 6.2. Here we select the 1 worker case at depth 0, i.e one sequential task, and choose the 120 worker depth that gives the best performance. The min–max scaling ranges are given as the ratio of the fastest 1 worker run to the slowest 120 worker run, and the fastest 120 worker run to the slowest 1 worker run respectively. That is, they represent the worst/best case scaling.

The Maximum Clique instances scale well, even with a low d_{cutoff} of 2, taking the performance from minutes to seconds. Given the low runtimes for the 120 worker case, it is unlikely that adding additional workers will improve performance by much (if not negatively affect performance due to overheads). For the brock400 series we see an example of a acceleration anomaly causing superlinear speedups. This is due to the speculative parallelism finding an improved bound earlier than a sequential search, reducing the overall workload.

As before, the Knapsack and Numerical Semigroups results show limited scaling, with Knapsack always showing a slowdown over the 1 worker case.

In general, SIP also performs well, however meshes-pat3-tar401 in particular shows poor scaling with low values of d_{cutoff} indicating low branching factors near the root of the search tree.

For TSP a high value of d_{cutoff} is required. As the branching factor for TSP is maximally the number of cities (i.e 35, 36, and 39) it is likely that the amount of work is not the limitation

Application	Instance	Sequential Runtime (s)	1 Worker Runtime (s)	Best d_{cutoff}	120 Worker Runtime (s)	Relative Speedup (min–max)
Maximum Clique	MANN_a45	212.47	202.51	5	2.97	68.09 (49.32–71.47)
	brock400_1	397.35	406.09	2	3.27	124.23 (113.86–132.83)
	brock400_3	231.97	235.51	2	1.90	124.15 (101.10–147.06)
	brock800_4	1,730.47	1,738.96	2	21.29	81.68 (68.38–95.50)
	p_hat700-3	1,471.64	1,495.52	2	16.24	92.08 (75.71–97.27)
	sanr400_0.7	107.48	109.52	2	1.40	78.12 (73.25–81.74)
TSP	rand_35_37662	2,029.22	2,038.97	8	46.51	43.84 (36.34–49.56)
	rand_36_46956	1,543.36	1,569.10	8	13.99	112.13 (71.61–121.15)
	rand_39_16423	2,477.41	2,519.47	8	37.53	67.14 (56.11–74.82)
Knapsack	knapPI_11_200_1000_27	145.40	152.39	8	152.94	1.00 (0.78–1.08)
	knapPI_13_200_1000_48	2,556.24	2,563.81	5	2,756.86	0.93 (0.76–0.97)
	knapPI_14_200_1000_69	587.45	609.38	6	650.66	0.94 (0.86–0.97)
SIP	g18-g106	1,587.07	1,602.12	1	5.53	289.92 (9.99–374.82)
	g34-g79	3,451.46	3,463.55	6	35.41	97.80 (92.34–98.49)
	g36-g107	1,512.89	1,527.72	4	16.94	90.18 (88.02–92.86)
	meshes-pat3-tar401	2,871.59	2,892.44	8	649.09	4.46 (4.44–4.47)
	si2_r01_m600.06	3,491.69	3,505.59	2	26.42	132.68 (38.55–373.81)
Numerical Semigroups	47.0	1,229.67	1,309.89	35	835.09	1.57 (1.55–1.61)
UTS	T1XL	522.16	527.74	8	5.32	99.16 (97.40–100.76)
	T1XXL	1,351.72	1,361.48	0	1,449.79	0.94 (0.94–0.94)
	T1XXL+	5,427.80	NaN	NaN	NaN	NaN (NaN–NaN)

Table 6.2: Relative Speedup of Depth-Bounded with best d_{cutoff}. Median runtime shown. Speedup relative to 1 worker case. NaN values represent timeout after 1 hour.

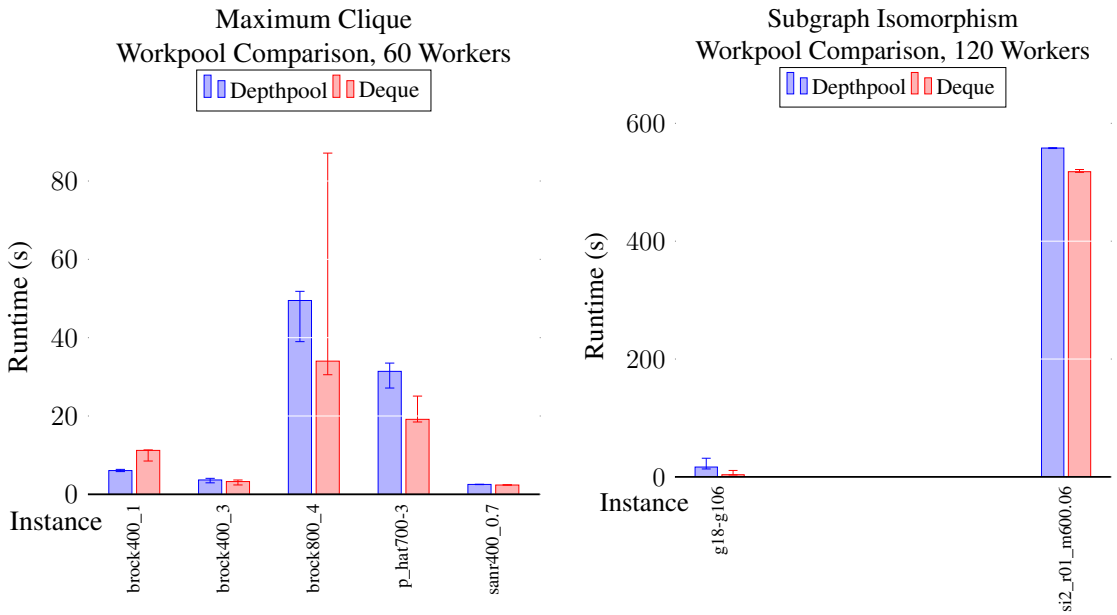


Figure 6.6: Impact of different workpools on Depth-Bounded. Median runtime shown. Error bars represent the minimum and maximum runtimes measured.

here, but that bounds propagation causes many of these tasks to be proved trivial (i.e. we have lots of uninteresting tasks).

The best d_{cutoff} setting for SIP is less consistent than the other benchmarks, e.g. 2 for Maximum Clique or 8 for TSP. The SIP instances come from a more varied set of instances likely giving them very different search tree shapes. In general we should not expect a single d_{cutoff} to work for all instances, as highlighted by the range of best d_{cutoff} values across all applications.

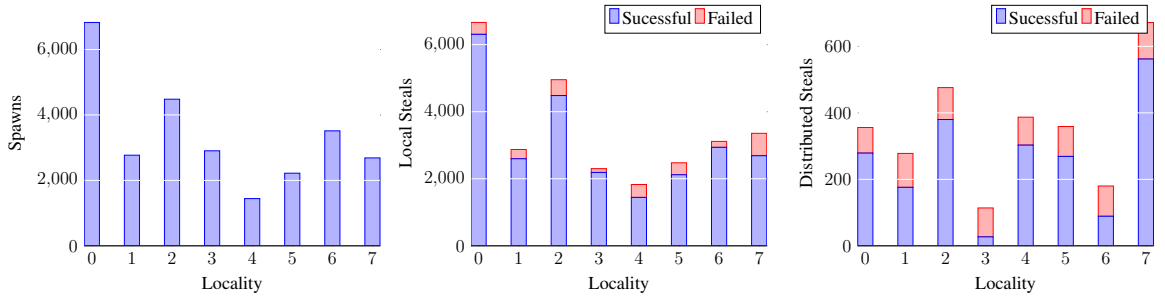
UTS scales well for T1XL, albeit requiring a deeper d_{cutoff}. This lower d_{cutoff} requirement is expected as the branching factor of each node is at most 4 (by definition).

6.5.1 Workpool Choice

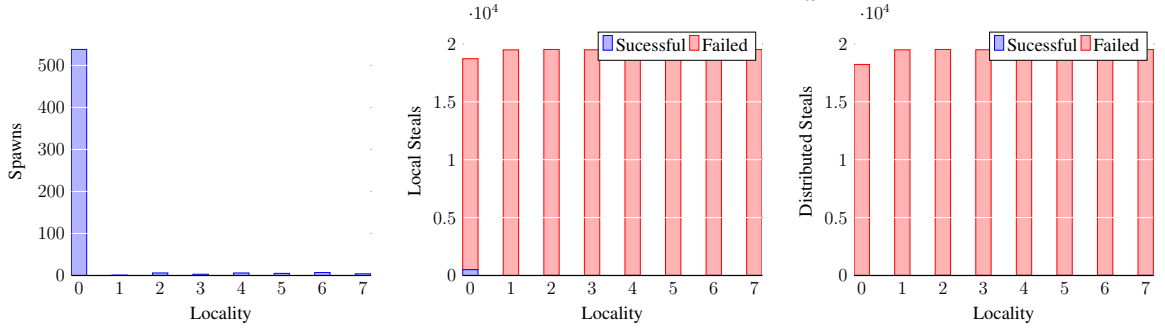
Section 4.4 discusses the issues with the common deque-based work-stealing, mainly that it often goes against the heuristic search order. In this experiment we consider the performance of Depth-Bounded as we change the type of workpool from depth-pool to deque. We consider the Maximum Clique and SIP implementations running on 60 and 120 workers respectively, with a d_{cutoff} of 2. Using 60 workers ensures the running time for clique is long enough to see any effects.

The effect of workpool choice on running time is shown in Figure 6.6.

For both Maximum Clique and SIP the workpool choice has limited effect on the overall runtime results. In general the depth-pool reduces performance, although not by much given



(a) brock400_1: work-stealing statistics, $d_{cutoff} = 2$.



(b) knapPI_14_200_1000_69: work-stealing statistics, $d_{cutoff} = 6$.

Figure 6.7: Work-stealing performance of Depth-Bounded.

the deque implementation is significantly more mature than the depth-pool.

brock800_4 shows particularly large range for deque based scheduling, likely due to differences in heuristic order. Depth-pool based scheduling appears to avoid this, giving more consistent results.

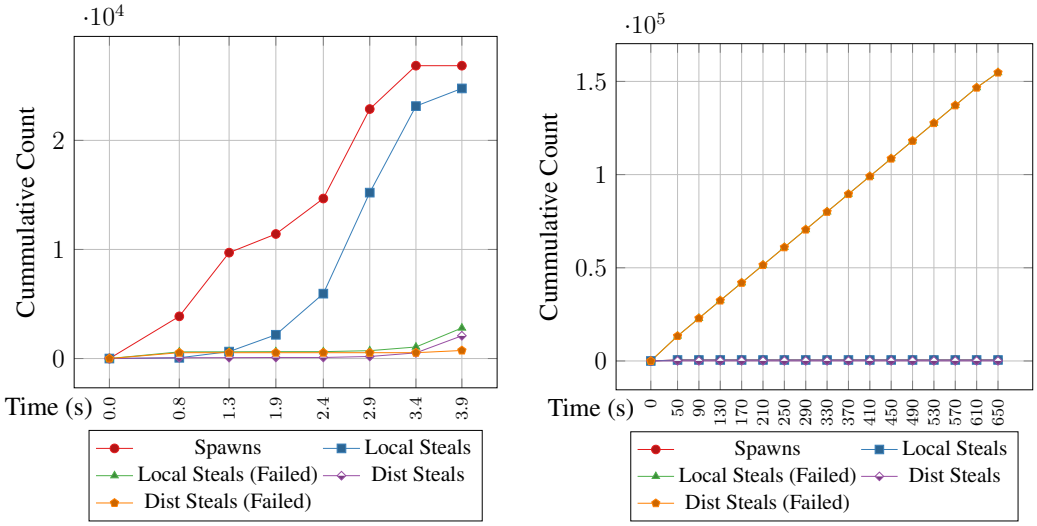
It may be that due to search heuristics generally performing badly near the root of the search tree [18] we see limited effect from the depth-pool at $d_{cutoff} = 2$. Trial runs at $d_{cutoff} = 4$ show similar behaviour as before with changing workpool having no clear effect on overall runtimes.

Given that performance is not significantly degraded, we choose to use the depth-pool when evaluating the skeletons as it is designed specifically for search.

6.5.2 Depth-Bounded Work-Stealing Performance

Figure 6.7 summarises work-stealing statistics, broken down by locality, for a successfully parallelised run, brock400_1, (Figure 6.7(a)) and an unsuccessfully parallelised run, knapPI_14_200_1000_69 (Figure 6.7(b)). In each case we use performance information from a single sample at the best d_{cutoff}.

Figure 6.7(a) shows that Depth-Bounded achieves a good work distribution. By spawning tasks as they are executed, instead of ahead of time, the tasks become spread across the



(a) brock400_1: work-stealing statistics over time.

(b) knapPI_14_200_1000_69: work-stealing statistics over time.

Figure 6.8: Work-stealing performance of Depth-Bounded over time.

localities with most reaching more than 2000 tasks. The high number of tasks ensures a large proportion of the steals occur locally, with limited distributed steals needing to occur (< 500). YewPar handles the large numbers of tasks efficiently, scheduling and running 26924 total tasks in 3.5s (7692 tasks per second).

Figure 6.7(b) tells a much different story. Here there is a limited number of spawns and of these almost all of them happen on a single locality. The large number of both local and distributed steals shows the system is starved of work. This is expected for Knapsack as most of the work is dominated by a few left-most branches. With no method of dynamically generating new tasks after d_{cutoff} is reached, the runtime is bounded by the long running tasks.

Figure 6.8 shows work-stealing statistics as a function of time for both brock400_1 and knapPI_14_200_1000_69. Figure 6.8(a) illustrates how the total spawned tasks gradually increases over the run rather than being generated upfront (as in many static approaches described in Section 2.4.1). The number of tasks waiting to run is given by the distance between the red spawn curve and the combined successful steals curves. This shows that the total final amount of tasks is never present in the system at once, keeping memory requirements low. In the middle of the search we have many more spawns than steals, showing there is plenty of work left in the system (i.e. starvation is avoided). As expected, failed steals increase towards the end of the search as work becomes sparse.

As expected given the poor scaling, knapPI_14_200_1000_69 (Figure 6.8(b)) is dominated by failed steals, shown by the linearly increasing failed steal line (both local and distributed). Spawns, local steals, and distributed steals, change very little over the full run.

6.5.3 Depth-Bounded Summary

Perhaps surprisingly, given the simplicity of Depth-Bounded work generation, it performs well for many of the case studies.

We have shown the overheads of Depth-Bounded can be low, particularly for small values of d_{cutoff} (Figure 6.3). Even though tasks are generated during search instead of upfront, large values of d_{cutoff} can lead to large increases in the numbers of tasks that are be difficult to manage (Figure 6.4).

When parallelism is introduced, Depth-Bounded can successfully reduce runtimes by an order of magnitude, with a maximum speedup of 122 on 120 workers (Table 6.2). Depth-Bounded works particularly well for Maximum Clique, TSP, and some SIP instances. It struggles to parallelise Knapsack, where a single branch dominates the workload, and Numerical Semigroups, where branching factors are often low near the root of the search. For UTS performance appears to be related to the instance, showing how the shape of the search tree plays a role in overall performance. A key disadvantage is the inability to dynamically split work if the current workpools are exhausted.

Automatically determining an optimal value for d_{cutoff} remains an open problem. For some applications a single choice of d_{cutoff} works well for most instances, e.g. 2 for Maximum Clique. However this is not always the case and, for example, SIP has no d_{cutoff} that works well for all instances.

Section 4.4 discusses the shortcomings of typical deque based work-stealing when applied to combinatorial search applications, and proposes a new structure, the depth-pool, that attempts to maintain heuristic orderings as much as possible while still allowing differences between remote and local steals. We have shown (Section 6.5.1) that the performance of the depth-pool is comparable to deque scheduling, allowing search to be performed in a more principled manner without significant overhead.

Finally, a deeper look into the work-stealing performance of Depth-Bounded (Section 6.5.2) shows Depth-Bounded achieves good load balance for Maxiumum Clique and highlights the lack of work issues present for Knapsack.

6.6 Stack-Stealing

Stack-Stealing (Section 4.3.4) allows idle workers to steal directly from the search tree of other, both local and remote, workers. The approach requires no parameter tuning from the user, instead the amount of parallelism is based on system properties, i.e. idle workers.

Table 6.3 shows the relative speedup of Stack-Stealing without chunking. Speedup ranges are given as in Section 6.5. Stack-Stealing works well in many cases, successfully bringing

Application	Instance	Sequential Runtime (s)	1 Worker Runtime (s)	120 Worker Runtime (s)	Relative Speedup (min–max)
Maximum Clique	MANN_a45	212.47	219.90	8.40	26.19 (26.11–26.86)
	brock400_1	397.35	483.65	6.47	74.80 (53.34–84.12)
	brock400_3	231.97	283.72	4.55	62.34 (57.43–87.01)
	brock800_4	1,730.47	1,991.11	74.53	26.72 (20.88–51.36)
	p_hat700-3	1,471.64	1,634.48	40.95	39.91 (37.65–49.50)
	sanr400_0.7	107.48	134.13	2.47	54.24 (53.20–57.18)
TSP	rand_35_37662	2,029.22	2,024.66	151.97	13.32 (11.66–23.60)
	rand_36_46956	1,543.36	1,561.49	35.82	43.60 (32.24–52.69)
	rand_39_16423	2,477.41	2,511.62	120.76	20.80 (16.66–24.56)
Knapsack	knapPI_11_200_1000_27	145.40	154.90	12.75	12.15 (0.90–13.00)
	knapPI_13_200_1000_48	2,556.24	2,689.08	2,822.27	0.95 (0.90–29.90)
	knapPI_14_200_1000_69	587.45	644.15	30.22	21.31 (0.89–35.21)
SIP	g18-g106	1,587.07	1,583.88	4.56	347.19 (272.11–353.87)
	g34-g79	3,451.46	3,379.29	51.36	65.79 (64.36–66.66)
	g36-g107	1,512.89	1,492.17	32.87	45.39 (35.88–50.61)
	meshes-pat3-tar401	2,871.59	2,837.36	90.43	31.38 (16.90–36.50)
	si2_r01_m600.06	3,491.69	3,399.52	7.71	440.87 (128.88–476.96)
Numerical Semigroups	47.0	1,229.67	1,888.23	46.57	40.54 (24.52–51.54)
UTS	T1XL	522.16	548.63	10.10	54.30 (37.77–58.02)
	T1XXL	1,351.72	1,411.49	21.10	66.89 (21.67–73.75)
	T1XXL+	5,427.80	NaN	100.31	NaN (NaN–NaN)

Table 6.3: Relative speedup of Stack-Stealing (without chunking). Median runtime shown. Speedup relative to 1 worker case. NaN values represent timeout after 1 hour.

runtime down from minutes to seconds. As with Depth-Bounded, may instances likely will not scale further given the low runtime for 120 workers.

On average Stack-Stealing performs particularly poorly for Knapsack, achieving limited improvements over the 1 worker case. However, the results have huge variance, e.g. in the best case KnapPI_13_200_1000_48 only takes 96s to solve, a speedup of almost 30. Again this is likely due to the strong search heuristic of Knapsack, forcing the solution deep into the leftmost branch. Stack-Stealing may perform many steals in non critical branches (i.e. those to the right) causing poor speedups, or get lucky and find the critical branch allowing redistribution of work, achieving much better runtimes.

For SIP, Stack-Stealing manages superlinear speedups in the two satisfiable cases. This is not true for Depth-Bounded, suggesting that the random work-stealing, or the work-pushed initial tasks of Stack-Stealing, is effective at adding diversity to the search, leading to finding a solution earlier. Interestingly, the standard deviations in the superlinear cases are relatively low (compared to the sequential runtime) which seems to imply that even though the steals are random, the critical branches are often found quickly.

6.6.1 Use of Chunking

An optimisation for Stack-Stealing is chunking, where multiple tasks are returned on a steal and buffered for later use. This allows for a fast steal path that does not interrupt a worker.

Figure 6.9 shows the effect of using chunking for each application using 120 workers. Results are reported as the median over 5 runs with error bars reporting the minimum and maximum runtime measured. Given the high standard deviation of Stack-Stealing results, it is unclear if chunking is advantageous.

For Maximum Clique chunking often results in a slowdown, particularly for larger instances. As chunking takes all tasks at the lowest possible depth, chunk size is related to the average branching factor of an instance e.g. brock800_4 has large branching factors near the root causing large chunk sizes. If there are not enough workers free to take this work, then we pay the price of spawning and scheduling tasks that they would have searched without overhead if chunking was disabled.

To investigate this, Figure 6.10 shows the distribution of stolen chunk sizes for a random set of instances across all applications. For most instances chunk sizes are often less than 200 tasks, with the exception of brock800_4 that commonly sees chunksizes of around 400 tasks. As we expect, for larger chunk sizes the total number of steals (from workers, not the chunk pool) are reduced.

SIP is particularly interesting, as no chunk sizes are greater than 38. This is due to low branching factors in SIP as the search space is quickly constrained. Even with this low chunk

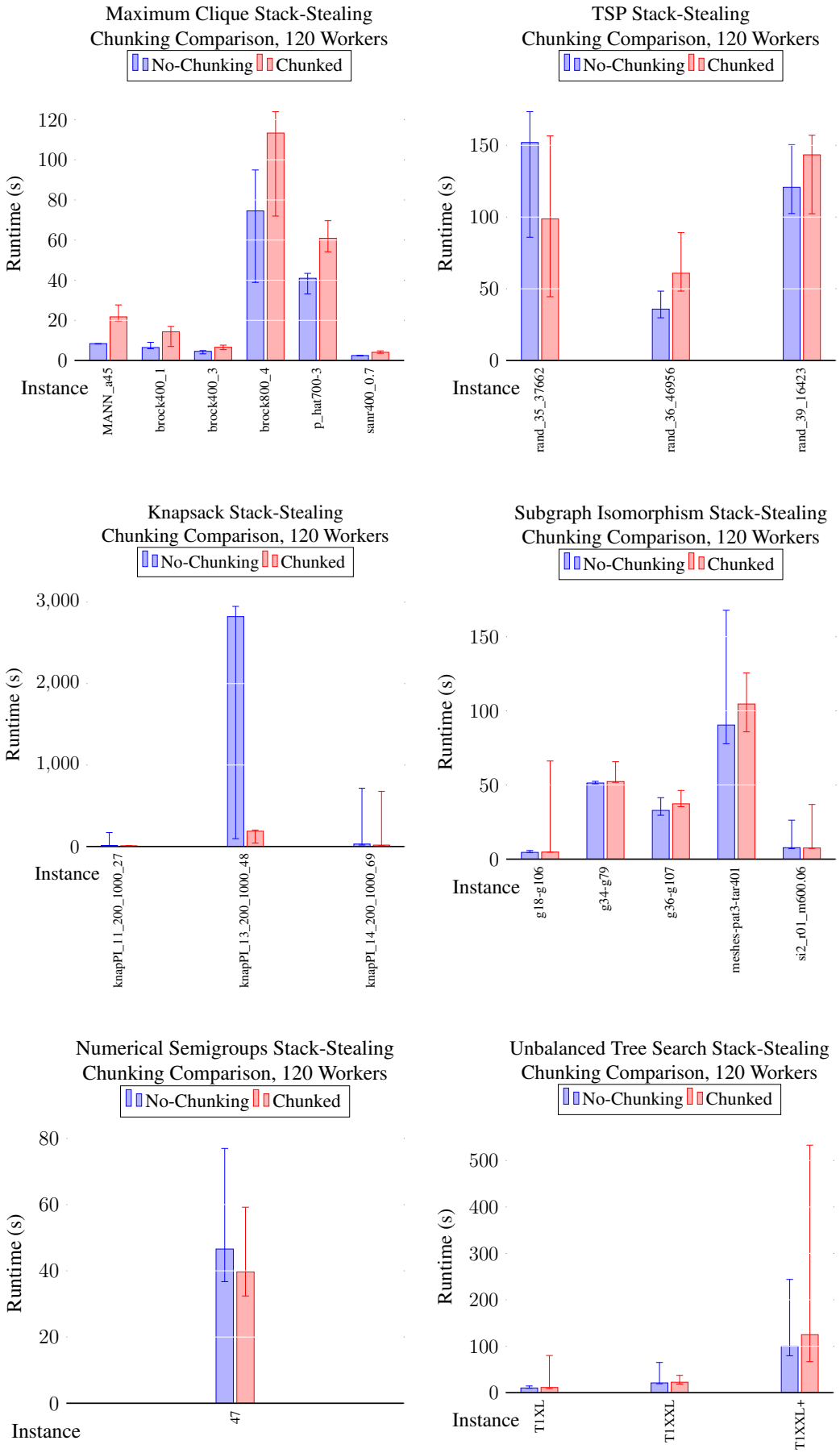


Figure 6.9: Effect of chunking on Stack-Stealing performance using 120 Workers. Error bars show minimum and maximum runtime.

size, it does not look like chunking is beneficial for meshes-pat3-tar-401. In general there seems to be little relationship between chunk size and running time.

6.6.2 Stack-Stealing Work-stealing Performance

Figure 6.11 shows the work-stealing performance of Stack-Stealing for a single sample of brock800_4 without chunking. As we expect, given that Stack-Stealing can perform steals from a worker at *any* depth, most of the local steals are successful.

Unlike Depth-Bounded, Stack-Stealing initiates distributed steals whenever there are no active local workers that are not currently being stolen from, rather than when all workers are fully idle. This accounts for the higher number of distributed steals. It is possible that this approach is too aggressive, and instead should ensure all workers are idle before attempting distributed steals.

Given the low rate of steals from locality 0, it is likely that many of the large tasks are held there. As Stack-Stealing initially distributes top level tasks in a round-robin fashion it is interesting to see the large tasks clustering in this way.

One difficulty analysing Stack-Stealing is that, due to the randomness in the work-stealing there is often a lot of variance in the runs (see Figure 6.9). Figure 6.12 shows the work-stealing performance for two different runs knapPI_14_200_1000_69. Figure 6.12(a) shows a particularly bad run where the work has clustered on locality 1 and all other localities are struggling to find it. Given the high number of steals this is surprising as we might expect, by the randomness of work-stealing, that eventually localities should find work (particularly given the large 715.2s runtime), and it is unclear why they do not. Figure 6.12(b) shows the expected behaviour of Stack-Stealing, all localities have enough work that idle workers tend to find work locally, and distributed steals are successful in finding more work as required.

Unfortunately, the Stack-stealing implementation currently interacts badly with the HPX performance counter system, making it difficult to determine how the work-stealing scheduler interacts over time (as in Figure 6.8). This makes it difficult to prove, for example, where the large steal counts occur in brock800_4 and why no work is found in the poor knapPI_14_-200_1000_69 example.

6.6.3 Stack-Stealing Summary

Stack-Stealing performs well for many instances, successfully reducing runtimes without any requirement to manually tune parallelism parameters. Due to the randomness of work-stealing, the standard deviation of the results is often high. This is particularly prevalent in Knapsack

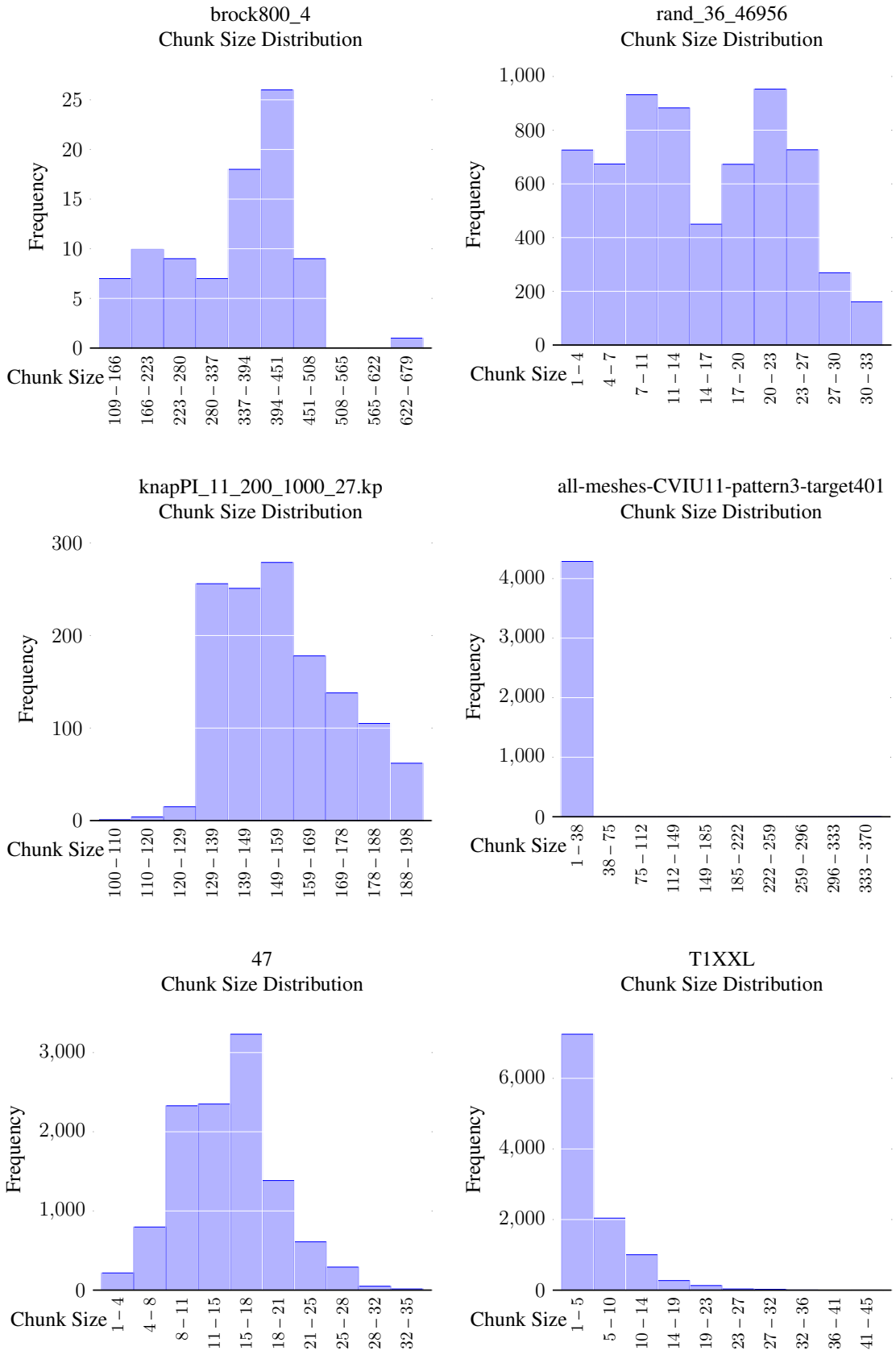


Figure 6.10: Chunk size distribution for Stack-Stealing. Note the differences in x-axis scale.

Figure 6.11: brock800_4: Stack-Stealing work-stealing Statistics (no chunking).

where one instance has an average speedup of 1.5, yet for a specific run achieves a 64 times speedup.

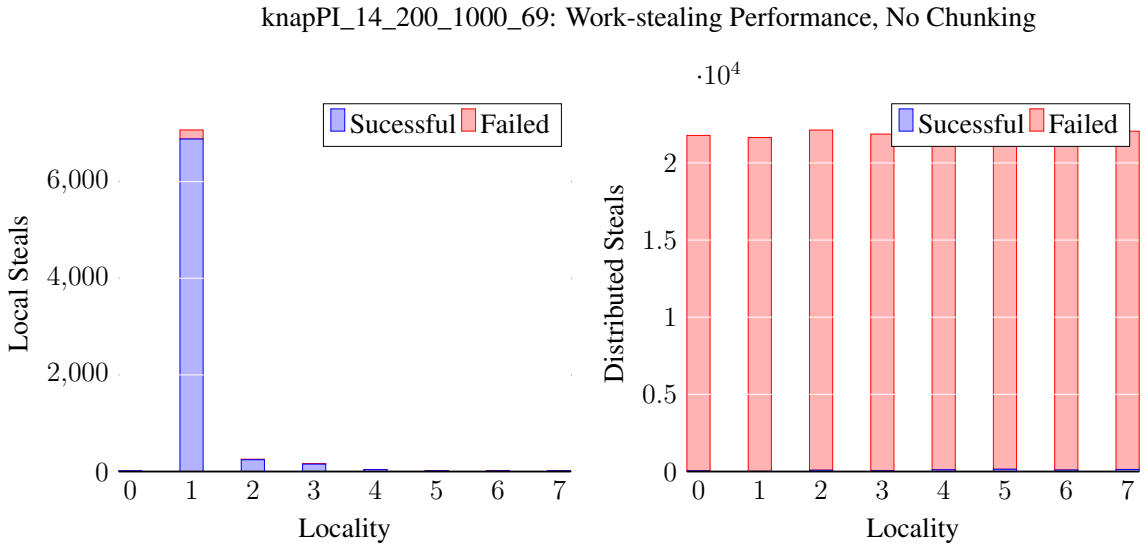
We have shown that the chunking optimisation does not work as well as expected (Section 6.6.1), often showing no significant improvement over the non-chunked version. The chunk sizes that are stolen differ on an instance by instance basis (related to the branching factor of the instance) and there does not appear to be a direct relationship between the number and size of chunks and associated runtime changes.

A deeper look into the work-stealing performance of Stack-Stealing (Section 6.6.2) shows Stack-Stealing often finds work locally, however, possibly due to allowing distributed steals when some workers are not fully idle, the number of distributed steals can be high. The variance in Knapsack results is highlighted by showing how the work tends to cluster on a single locality. It is not clear why the distributed steals are unable to find this in order to further split the workload.

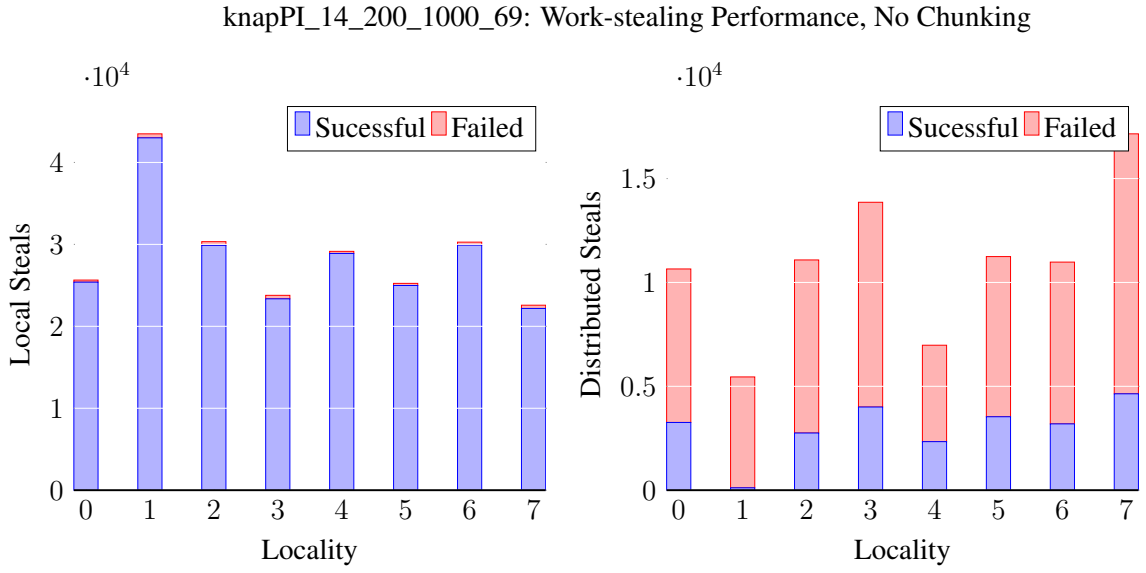
Stack-stealing currently interacts poorly with the HPX schedulers, affecting printing of performance counters, as well as shutdown. The root cause of these problems is unknown, however Stack-Stealing both consistently gives the correct answer and achieves parallel performance.

6.7 Budget

Budget (Section 4.3.6) performs search until a specified backtrack budget is met, at which point all nodes at the lowest depth (for the current worker) are converted to tasks. A disadvantage of this search coordination is the requirement to choose a suitable budget.



(a) Poor run: 715.2s



(b) Good run: 18.4s

Figure 6.12: knapPI_14_200_1000_69: Stack-Stealing work-stealing performance (no chunking).

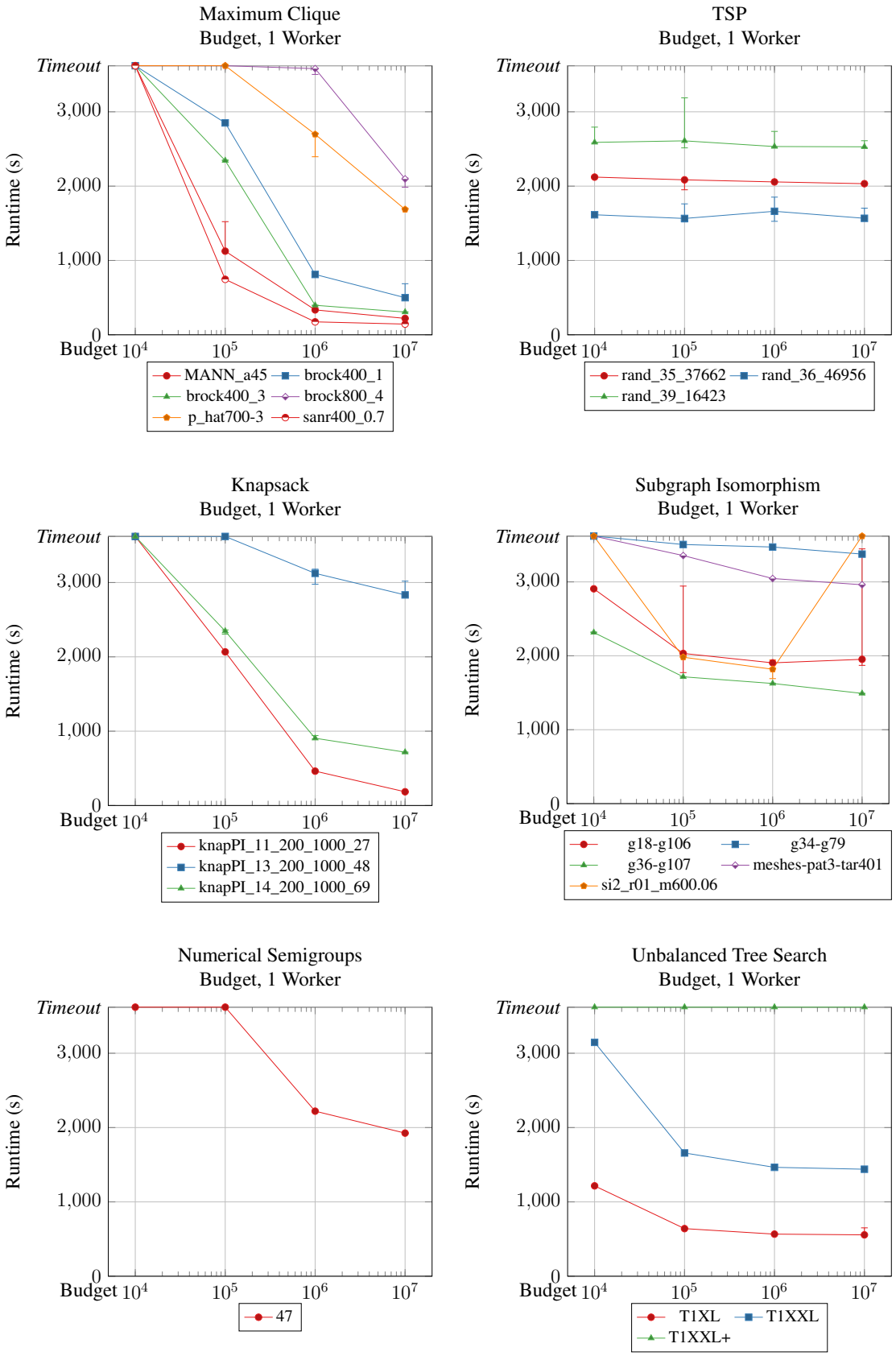


Figure 6.13: Impact of Budget on 1 worker runtimes. Error bars show minimum and maximum runtime.

Figure 6.13 shows the median 1 worker runtime results as the budget parameter is varied between 10^4 and 10^7. These parameters were chosen based on initial experimentation showing that they allow a range of application runtime behaviour trends to be observed, without requiring large amounts of compute resource. We expect runtime to decrease as budget increases as less tasks are being spawned, reducing the total scheduling overheads. In general this trend is followed, with many instances, particularly Maximum Clique, Knapsack and Numerical Semigroups struggling to run in less than one hour with $budget = 10^4$. TSP does not follow this trend, instead it shows very little changes with the budget. This looks to be caused by low branching factors causing limited numbers of tasks to be generated even at low budgets (around 100,000 maximally compared with 2,100,000 for knapsack).

We expect relatively low variance in the results as, for the one worker case, task ordering should be consistent so long as the budget is fixed. g18-g106, a satisfiable SIP instance, surprisingly does not follow this trend, showing particularly high variance for $budget = 10^7$. The number of tasks generated is the same in each run and it is not clear what causes this effect; one possibility is that the (global) early termination flag, that is set in a second thread, is somehow being delayed[6]. TSP likewise has a high variance of unknown cause.

Figure 6.14 shows the median 120 worker runtime results as the budget is varied between 10^4 and 10^7. For large Maximum Clique instances, Knapsack, Numerical Semigroups, and UTS, we often see high runtime for low budget values. This is explained by the large increase in total spawned tasks at these budgets (similar to Figure 6.4). For example, brock800_4 has a mean number of tasks of 9,471,656 for $budget = 10^4$ compared with 1,001,047 for $budget = 10^5$, almost a 10 times increase. While this mirrors the 10 times increase in runtime, the ratio of task increases and runtime increases is not always equal. Knapsack and Numerical Semigroups show similarly large task counts at low budgets.

For TSP and SIP low budget settings perform well. At high budget settings low numbers of tasks are generated, e.g. an average 177 tasks for rand_39_16423, making it likely that the system starves.

Perhaps surprisingly, for many cases $budget = 10^5$ appears to be a good, yet not necessarily the best, setting. This is remarkable given the differences in the applications and seems to represent a good tradeoff between keeping the processors busy and not overloading the scheduler system.

Table 6.4 shows the speedup of the 120 workers over the best 1 worker runtime. Speedup ranges are given as in Section 6.5. It is not the case the 1 worker runtime is fully sequential, as even with high budgets tasks may be spawned.

As with the other skeletons, Budget successfully brings runtimes down to a few seconds/min-

[6]A free thread is allocated during the run which should allow this update task to be scheduled in parallel with the search; even in the one worker case.

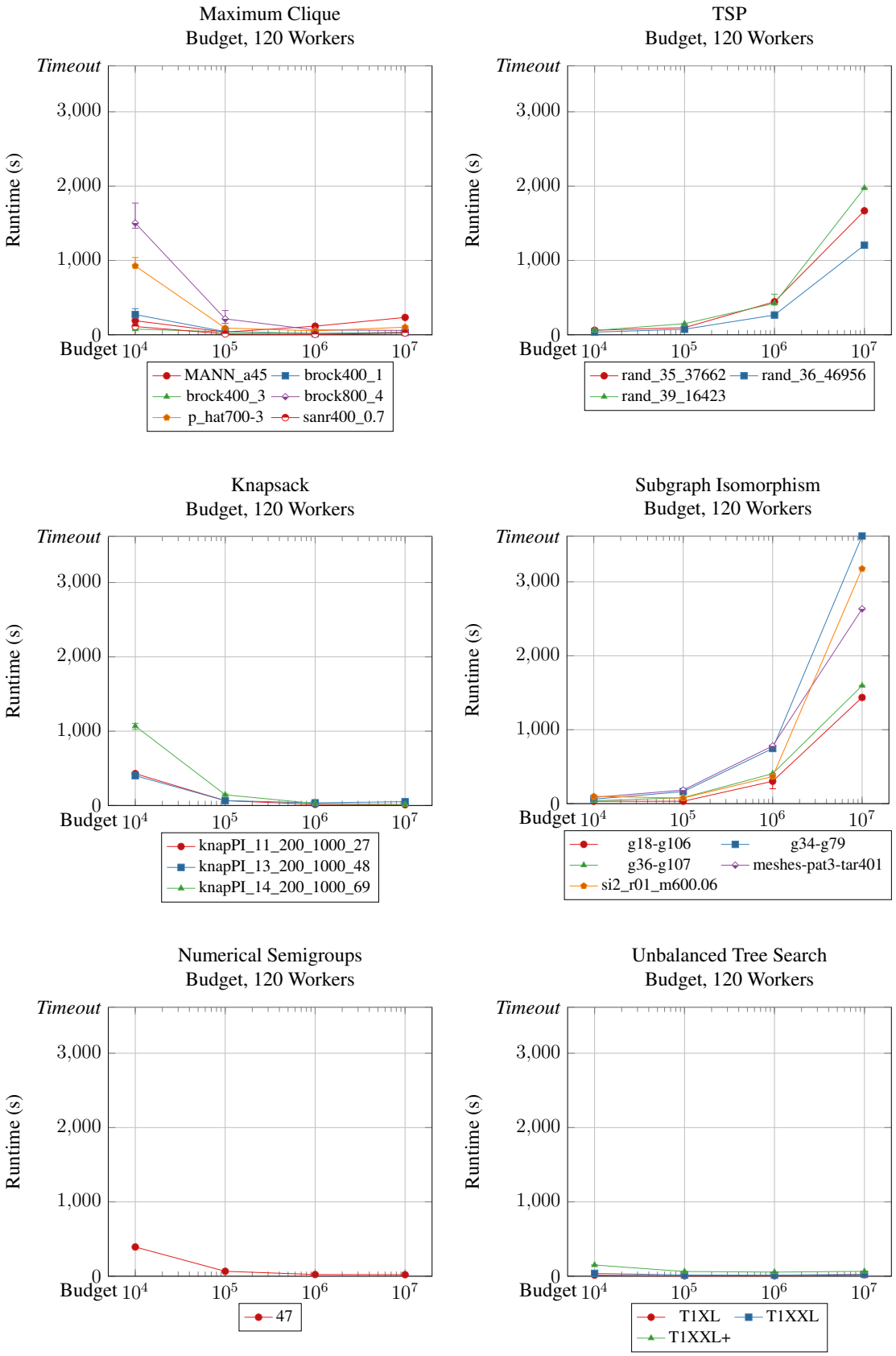
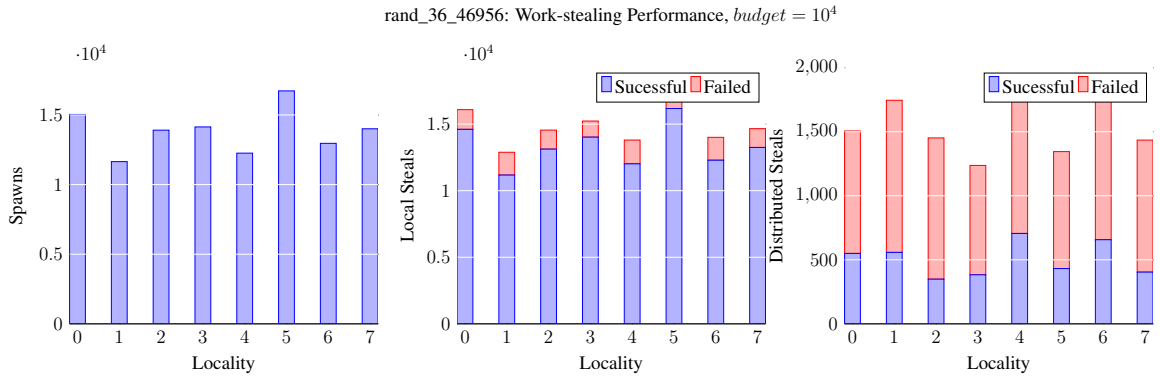


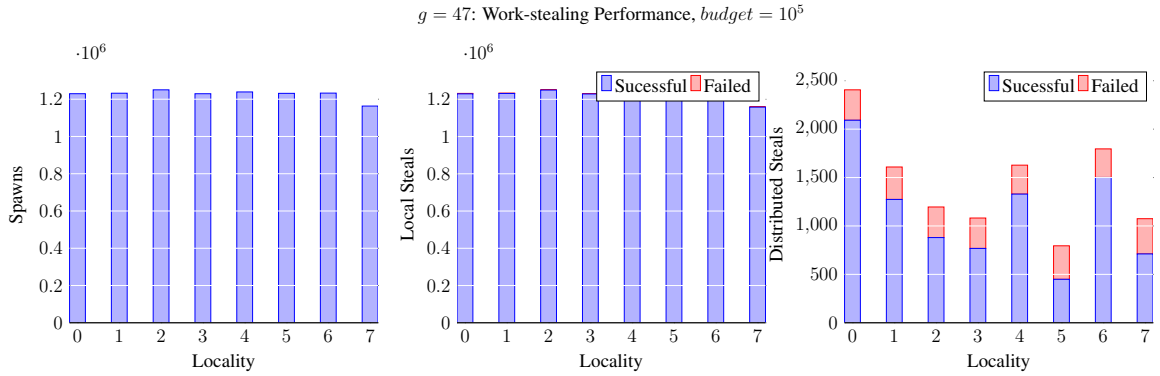
Figure 6.14: Impact of Budget on 120 worker runtimes. Error bars show minimum and maximum runtime.

Application	Instance	Sequential Backtracks/s	Sequential Runtime (s)	1 Worker Budget	1 Worker Runtime (s)	120 Worker Budget	120 Worker Runtime (s)	Relative Speedup (min–max)
Maximum Clique	MANN_a45	$7.0 \cdot 10^{11}$	212.47	10^7	220.78	10^5	36.32	6.08 (5.59–6.20)
	brock400_1	$3.4 \cdot 10^{11}$	397.35	10^7	499.32	10^6	18.32	27.26 (25.81–48.14)
	brock400_3	$5.9 \cdot 10^{11}$	231.97	10^7	305.85	10^6	14.6	20.94 (16.91–32.79)
	brock800_4	$8.0 \cdot 10^{10}$	1,730.47	10^7	2,097.12	10^7	58.82	35.65 (23.59–43.62)
	p_hat700-3	$9.4 \cdot 10^{10}$	1,471.64	10^7	1,687.13	10^6	53.23	31.70 (31.22–36.57)
	sanr400_0.7	$1.2 \cdot 10^{12}$	107.48	10^7	144.06	10^6	5.44	26.46 (26.01–27.19)
TSP	rand_35_37662	$1.2 \cdot 10^4$	2,029.22	10^7	2,032.93	10^4	60.68	33.50 (23.59–37.25)
	rand_36_46956	$9.5 \cdot 10^3$	1,543.36	10^5	1,565.13	10^4	35.52	44.06 (33.68–51.93)
	rand_39_16423	$6.1 \cdot 10^3$	2,477.41	10^7	2,529.17	10^4	58.22	43.44 (41.11–47.01)
Knapsack	knapPI_13_200_1000_48	$5.4 \cdot 10^{10}$	2,556.24	10^7	2,833.35	10^6	35.56	79.67 (77.35–85.61)
	knapPI_14_200_1000_69	$2.2 \cdot 10^{11}$	587.45	10^7	717.68	10^7	11.5	62.40 (43.35–135.85)
	g18-g106	$9.1 \cdot 10^{10}$	1,587.07	10^6	1,904.73	10^4	31.02	61.40 (59.11–62.77)
SIP	g34-g79	$4.2 \cdot 10^{10}$	3,451.46	10^7	3,376.19	10^4	60.12	56.16 (54.05–57.92)
	g36-g107	$9.6 \cdot 10^{10}$	1,512.89	10^7	1,491.52	10^4	38.12	39.12 (38.95–41.45)
	meshes-pat3-tar401	$5.1 \cdot 10^{10}$	2,871.59	10^7	2,961.42	10^4	83.85	35.32 (34.55–36.28)
	si2_r01_m600.06	$4.2 \cdot 10^{10}$	3,491.69	10^6	1,817.98	10^5	74.93	24.26 (20.58–30.66)
	47	$9.0 \cdot 10^{10}$	1,229.67	10^7	1,925.42	10^7	20.65	93.25 (92.53–94.50)
Numerical Semigroups	T1XL	$3.1 \cdot 10^6$	522.16	10^7	556.63	10^5	6.4	86.97 (86.83–102.12)
UTS	T1XXL	$3.1 \cdot 10^6$	1,351.72	10^7	1,439.49	10^6	14.95	96.27 (95.07–97.98)
	T1XXL+	$3.1 \cdot 10^6$	5,427.80	NaN	NaN	10^6	55.52	NaN (NaN–NaN)

Table 6.4: Budget speedup over best 1 worker run. Median runtime shown. Speedup relative to 1 worker case. NaN values represent timeout after 1 hour.



(a) rand_36_46946: work-stealing statistics.



(b) Numerical Semigroups: work-stealing statistics.

Figure 6.15: Work-stealing performance of Budget.

utes. The variance of the results tends to be low (even in the cases with large one worker variances).

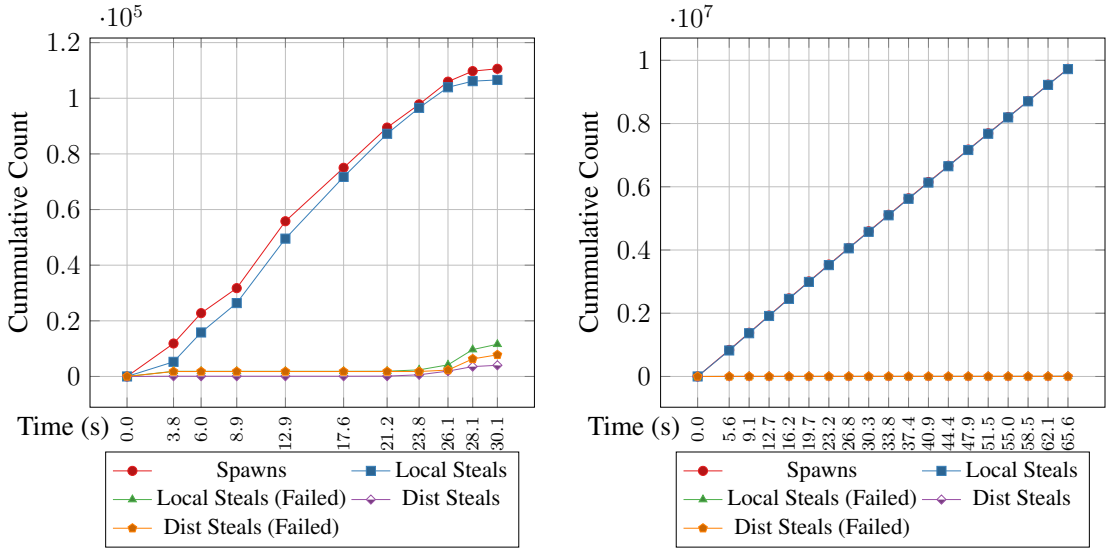
Budget does not perform poorly for any particular type of application, and shows stand out performance for Knapsack that has struggled to scale using the other skeletons. This is likely caused by the fact that Knapsack solutions are often in the leftmost branch and, by avoiding spawns until we predict a sub-tree has useful work, Budget avoids the overheads of processing useless trees.

Although Figure 6.14 showed $budget = 10^5$ to be a good across many applications, we see that achieving the best runtimes often require a different setting. It is also unlikely to continue to be the best budget at all scales.

6.7.1 Budget Work-Stealing Performance

Figure 6.15(a) shows the work-stealing statistics for the TSP instance rand_36_46946. In general the load is well balanced with most localities generating similar amounts of work, and a majority of local steals being successful.

Figure 6.15(b) shows the work-stealing statistics for Numerical Semigroups ($g = 47$). The



(a) rand36__46946: work-stealing statistics over time.

(b) Numerical Semigroups $g = 47$: Work-stealing statistics over time.

Figure 6.16: Work-stealing performance of Budget over time.

statistics are similar to that of rand_36_46946 with almost equal numbers of tasks being generated on all localities, and almost all steals being completed locally. The number of distributed steals are orders of magnitude less than those occurring locally and are likely occurring only at the start/end of search when work is sparse. YewPar itself has no issues with handling the large number of spawns.

The work-stealing statistics over time, combined over all localities, is shown in Figure 6.16. Statistics for rand_36_46946 are shown in Figure 6.16(a) and shows that most of the failed steals seen in Figure 6.15(a) occur near the end of search, as expected. Throughout the search the number of spawns and number of steals closely track each other, showing that very few tasks are actually being stored in the workpools for any length of time. Tasks look to be generated in the system at a steady rate.

Figure 6.16(b) tells a similar story, with spawns and steals tracking each other almost exactly; as shown by the linearly increasing local steal line (where spawns exactly match the local steal line). As shown by the near constant number of failed steals, there is always plenty of parallelism available for the entirety of the run.

6.7.2 Budget Summary

Budget performs well for almost all instances, including Knapsack where the other search coordinations struggle to see an improvement.

One difficulty of using Budget is there requirement to choose a suitable backtracking budget for each instance/application. For many applications $budget = 10^5$ appears to be a good, yet

not best, choice (Figure 6.14).

A deeper look into the work-stealing performance (Section 6.7.1) shows Budget maintains a good load balance, often generating similar numbers of tasks on each locality. For much of the run tasks are introduced into the system at a constant rate, providing enough local parallelism to avoid large numbers of distributed steals.

6.8 Skeleton Comparison

The previous section have focused on the performance of particular skeletons. In this section we compare and contrast the performance of the skeletons.

Figure 6.17 compares the performance of the three parallel skeletons on 120 workers. Runtimes are given for the best set of tuning parameters, i.e. d_{cutoff}, $budget$ and use of chunking, for each instance.

Maximum Clique performs well for all skeletons with all instances requiring less than two minutes to run. Depth-Bounded outperforms the other skeletons in all cases. Stack-Stealing tends to perform better than Budget, particularly for the smaller instances, e.g. brock400s.

Travelling Salesperson also performs well for all skeletons, however this time Budget outperforms Stack-Stealing. Stack-Stealing appears to be struggling to locate useful pieces of work, while Budget, by only generating expected large tasks, locates promising tasks easier.

For Knapsack, Budget significantly outshines the other skeletons that struggle to perform, particularly for the hard knapPI_13_200_1000_48 instance. Knapsack has a particularly poor distribution of non-trivial tasks and Budget, by only generating tasks it knows to be hard, responds well to this.

It is not obvious the best skeleton to choose for SIP. Depth-Bounded performs particularly poorly for meshes-pat3-tar401, yet redeems itself in the other cases.

Numerical Semigroups performs particularly poorly for Depth-Bounded, likely due to the limited branching factors at the top levels. Both Stack-Stealing and Budget, with their more dynamic nature, perform well with Budget having a slight edge over Stack-Stealing.

UTS results are similar to those of Numerical Semigroups with the Depth-Bounded skeleton failing to even run for T1XXL+ in less than an hour. Budget has a slight edge over Stack-Stealing in this case. It is possible that Stack-Stealing is stealing too low in the tree while Budget avoids this situation by generating work in a top down manner.

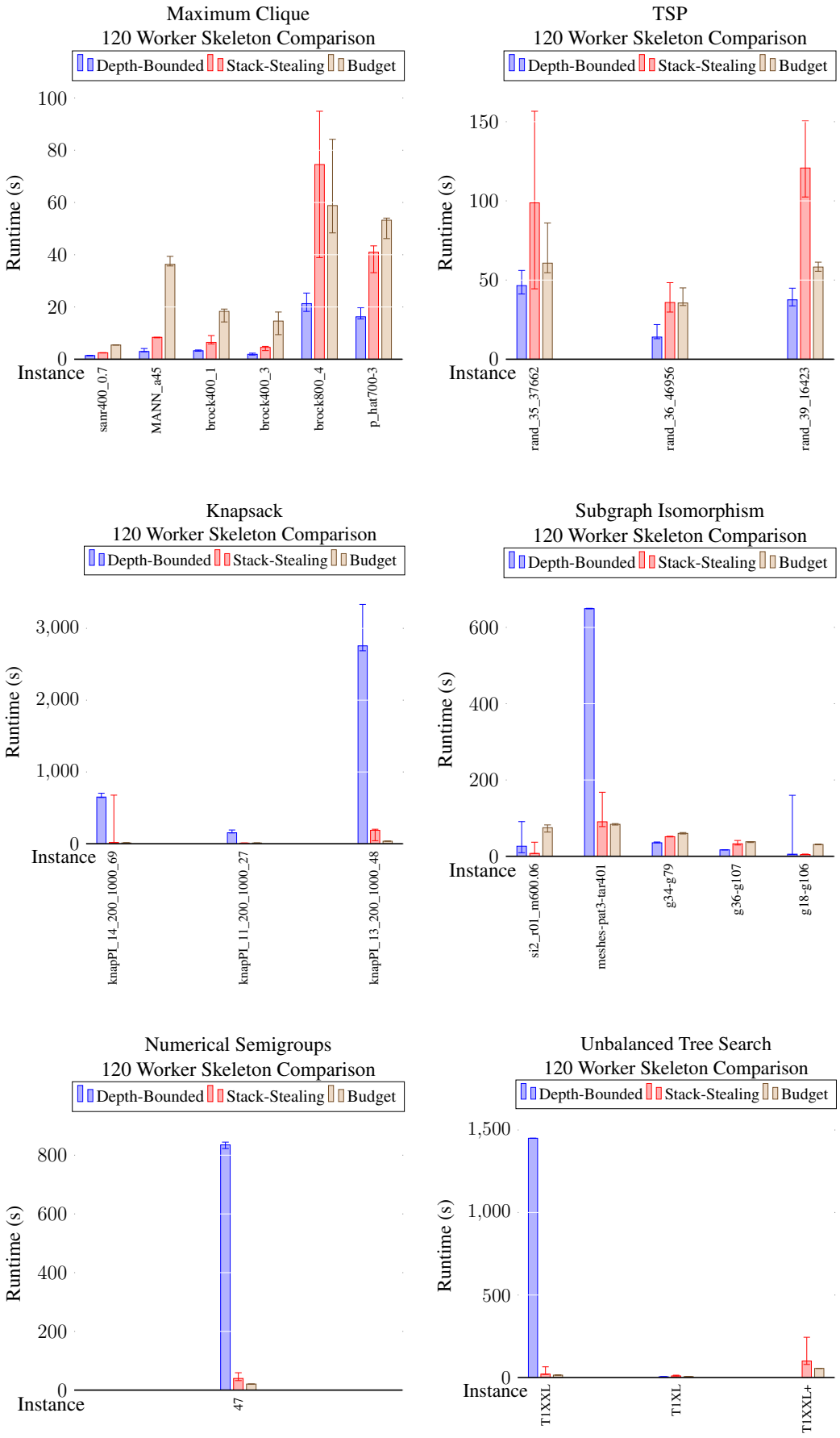


Figure 6.17: 120 worker Skeleton comparison. Lower is better. Error bars show minimum and maximum runtime.

Applications	Skeleton	Worst Scaling	Random Scaling	Best Scaling
Maximum Clique	**Depth-Bounded**	0.89	14.04	**91.74**
	Stack-Stealing	21.27	28.43	37.67
	Budget	1.38	7.58	17.84
TSP	**Depth-Bounded**	3.99	38.86	**68.19**
	Stack-Stealing	18.01	20.79	26.27
	Budget	1.25	4.89	39.52
Knapsack	Depth-Bounded	0.81	0.87	0.92
	Stack-Stealing	5.84	5.84	19.17
	Budget	1.06	8.75	**36.85**
SIP	Depth-Bounded	3.92	33.15	68.04
	Stack-Stealing	101.92	105.54	**109.61**
	Budget	1.42	7.7	45.05
Numerical Semigroups	Depth-Bounded	0.87	0.87	1.47
	Stack-Stealing	26.37	30.95	30.95
	Budget	3.12	3.12	**59.48**
UTS	Depth-Bounded	1.48	8.72	9.56
	Stack-Stealing	52.68	57.52	57.52
	Budget	29.81	85.85	**85.85**
All Applications	Depth-Bounded	1.67	8.69	26.51
	Stack-Stealing	27.99	34.21	**43.51**
	Budget	1.87	8.99	35.11

Table 6.5: Geometric mean scaling for 120 workers relative to Sequential.

6.8.1 Cumulative Statistics

Figure 6.17 showed that there is not always a clear best choice for skeleton in each case. To generalise the results, Table 6.5 gives the geometric mean scaling over all instances relative to the Sequential skeleton. Any instance that does not have scaling data available for each skeleton (including Sequential) is excluded from analysis.

One difficulty in comparing the results is that while, for example, Depth-Bounded works well for Maximum Clique, it also required experimentation to find a suitable d_{cutoff}. To see the effects of choosing poor tuning values, Table 6.5 also shows the scaling if the worst parameters are chosen[7], and the mean scaling for a randomly sampled set of parameter values.

There is no skeleton performs best for all cases. Depth-Bounded appears to work well for optimisation problems when there is plenty of work (i.e. Maximum Clique and TSP). Knapsack struggles with static work generation and improves significantly with Stack-Stealing and Budget. Stack-Stealing works well for SIP, giving an average of over 50% efficiency even

[7]We exclude $d_{cutoff} = 0$ as this generates no parallelism.

with the worst (chunking) parameter settings. Budget appears to be a good choice for the enumeration problems.

Stack-Stealing performs best on average and is a good choice when it is not clear the best set of tuning parameters to choose. Budget likewise performs well for most instances if the best Budget choice is available. Depth-Bounded, without the ability to dynamically adapt to instances, performs worst on average overall.

Importantly these results do not advocate Stack-Stealing as the single best skeleton in all circumstances, only that on average, it performs well for most instances. Specific applications work well for specific skeletons, e.g. Depth-Bounded for Maximum Clique.

6.9 Scalability

So far we have only considered instances with a sequential runtime of less than an hour, with parallelism often reducing this runtime to a number of seconds. In this section we consider both larger instances, those taking many hours to run sequentially, and scalability on up to 255 workers.

We consider a case study from each of the search types. Maximum Clique for optimisation, the Finite Geometry case study (Section 5.2.2.2) for decision, and Numerical Semigroups as an enumeration case study. Skeleton parameters are chosen based on the results of the skeleton studies (Section 6.5, Section 6.6 and Section 6.7), with some variation to account for the larger instances e.g. A larger budget to ensure we do not generate too many tasks. Due to the increased sequential runtimes, we report scaling relative to a single 15 worker locality (using the same skeleton).

Figure 6.18 shows the scaling performance of the Maximum Clique three brock800 instances that require more than one hour to run sequentially. The results mirror those of Figure 6.17 with Depth-Bounded being a clear best choice for Maximum Clique. brock800_3 shows superlinear scaling behaviour for Depth-Bounded, however not with the other skeletons, indicating that the dynamic nature of the other skeletons is stopping a useful branch from being explored early.

The interaction between Stack-Stealing and Budget is interesting. For low locality counts Budget scales much faster than Stack-Stealing. After around 8 localities are added, Stack-Stealing then begins to scale better than Budget. It is not clear exactly what causes this effect. One suggestion is that at low scale the ability for Budget to generate *good* tasks dominates. As we increase the scale the budget setting may be too low causing starvation. By adapting at runtime Stack-Stealing continues to scale.

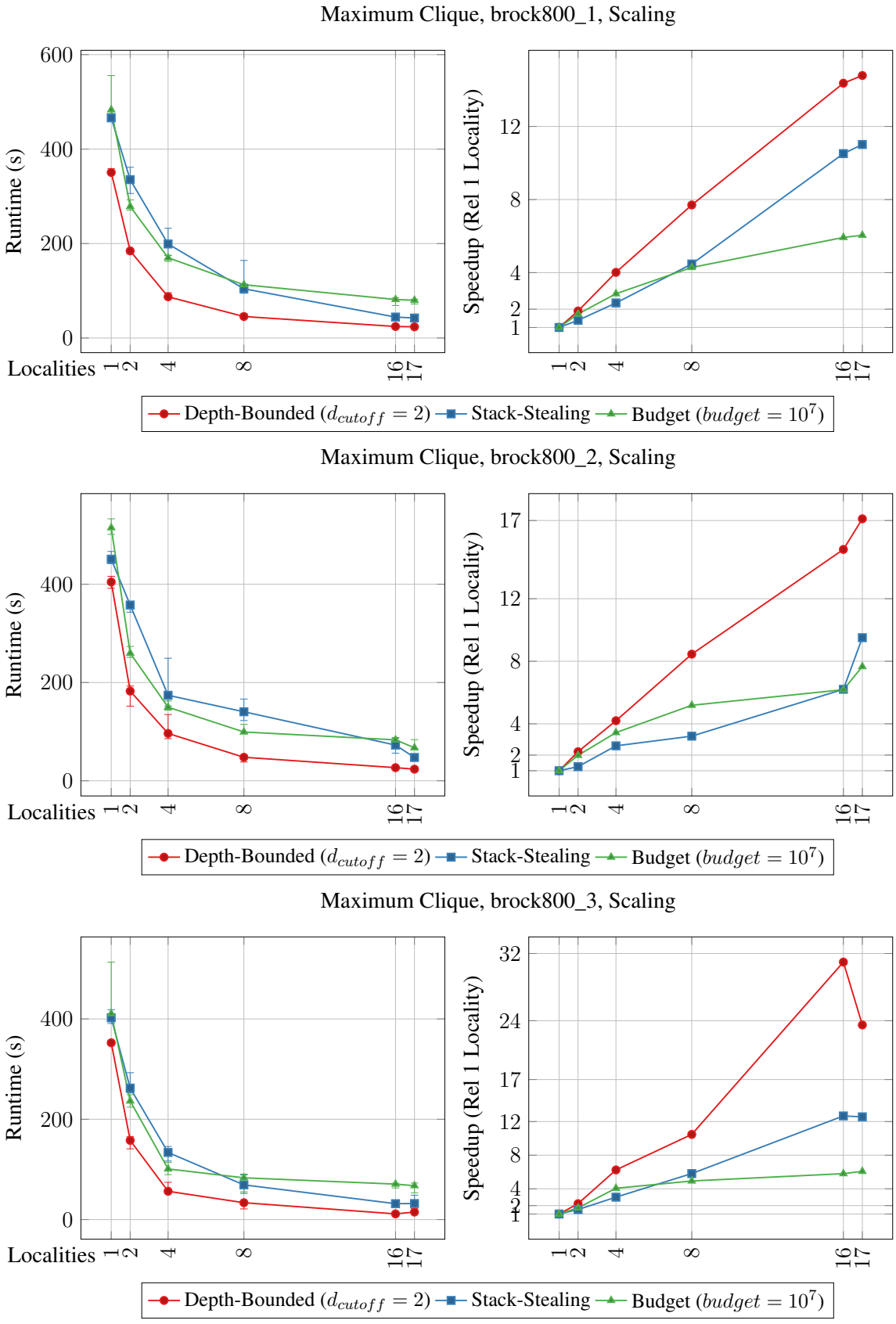


Figure 6.18: Scaling performance of Maximum Clique, large instances.

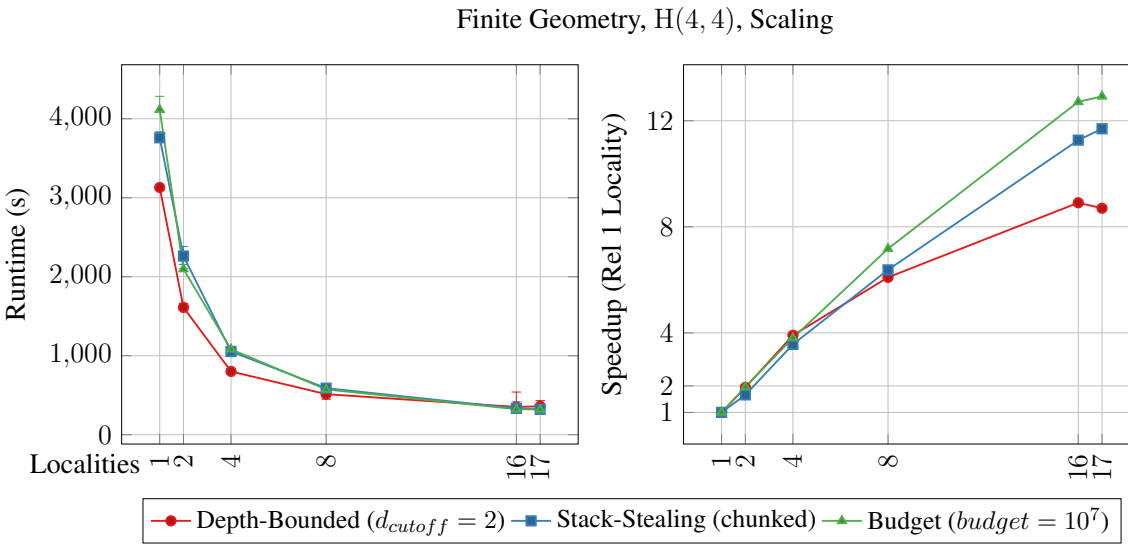


Figure 6.19: Scaling performance of Finite Geometry.

Figure 6.19 shows the scaling performance when searching for a spread in $\mathrm{H}(4, 2^2)$ which requires just under 11 hours to run using the Sequential skeleton[8]. As no spread exists, the search performs a full enumeration of $\mathrm{H}(4, 2^2)$; a fixed workload.

We observe the same disparity in the one locality runtimes as before. Perhaps surprisingly, given that the finite geometry search uses the same search code as Maximum Clique, the Depth-Bounded skeleton does not perform as well as in the optimisation case. In this case Budget, that often performed worst in the optimisation case, gives the best overall scaling, with Stack-Stealing a close second. In terms of absolute running times, both Stack-Stealing and Budget give a runtime of 315s while Depth-Bounded lags behind slightly at 373s.

Section 6.9 shows the scaling performance of Numerical Semigroups when counting the total nodes up to genus 50. As we saw in Figure 6.17, Budget performs particularly well for the Numerical Semigroups problem. Depth-Bounded ($d_{cutoff} = 40$) performs very poorly and times out (after 30minutes) regardless of the number of localities used. Stack-Stealing scales particularly slowly, often with < 50% efficiency.

Section 6.9 shows the mean efficiency (relative to one locality) the skeletons over all instance and scales. Except for Numerical Semigroups we see an average efficiency of greater than 60% in all cases; a good efficiency considering the irregularities of search. Maximum Clique obtains superlinear efficiencies in the case of Depth-Bounded due to superlinear scaling effects.

Overall we see that YewPar can scale to 255 workers enabling it to solve large, multi-hour sequential, instances in a few minutes.

[8] A (previously reported [5]) prototype version of YewPar, without skeletons, managed to solve this instance in around 4 hours. The difference in runtime is explained by a) use of a larger node structure (14 word bitsets instead of 8 words) and b) the additional copies required by the Node Generator API as discussed in Section 6.3.

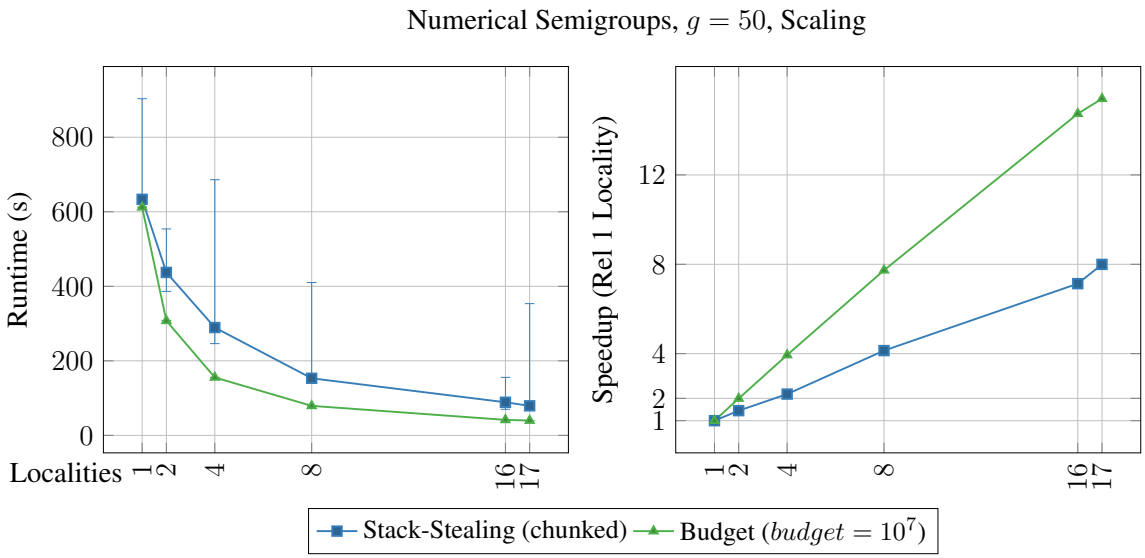


Figure 6.20: Scaling performance of Numerical Semigroups, large instance.

Applications	Skeleton	Average Efficiency (%)
Maximum Clique	Budget	60.95
	Stack-Stealing	63.2
	Depth-Bounded	112.74
Finite Geometry	Depth-Bounded	74.45
	Stack-Stealing	78.07
	Budget	87.48
Numerical Semigroups	Stack-Stealing	53.8
	Budget	95.46

Table 6.6: Geometric mean efficiency over all instances and scales.

Skeleton	Instances Solved	Mean Runtime (ms)
Depth-Bounded	177	144.29
Stack-Stealing	177	720.62
Budget	161	3,748.8

Table 6.7: Instances solved in less than 5 minutes on 255 workers when searching for spreads in $\mathrm{H}(4, 3^2)$. $d_{cutoff} = 2$, no chunking, $budget = 10^7$.

6.9.1 Finite Geometry: $\mathrm{H}(4, 3^2)$

As a final experiment we attempt to find spreads in the larger finite geometry $\mathrm{H}(4, 3^2)$. This instance is significantly harder than $\mathrm{H}(4, 2^2)$ with an expected sequential runtime of many days (likely months).

To solve this instance we use symmetry breaking via orbital branching (described in Section 5.2.2.2) until a depth of 4 to split the search into multiple smaller searches. There are 469 smaller searches in total.

We report the number of instances each skeleton can solve in 5 minutes or less using 255 workers in Table 6.7. Mean runtimes are calculated only for instances that are solved by all skeletons. With both Stack-Stealing and Depth-Bounded we can solve 37% of instances in less than five minutes and 34% of instances with Budget. No instance managed to find a spread.

Perhaps surprisingly, given that the Budget skeleton scales the best for $\mathrm{H}(4, 2^2)$ it performs worst in this case. It is not clear why but, as we have seen (Section 6.7), it could be down to a poor choice of $budget$. There seems to be little difference in the (mean) performance of Depth-Bounded and Stack-Stealing given that Stack-Stealing tends to have additional overheads in the one worker case.

Although there is a 5 minute timeout, the average runtime is less than 5 seconds. This indicates that the solved instances are likely to be the trivial region of search and much higher timeouts will be required to fully search this geometry.

6.10 Generality and Ease of Use

The previous sections provide empirical analysis of the skeletons. Before ending this chapter, we briefly comment on the generality and ease of use of YewPar.

We have shown generality by implementing 7 different case studies, featuring a mix of enumeration, decision and optimisation search types, as shown in Table 6.8. Clique search shows how the *same* node generator can be used for two different types of search without

Node Generator	Enumeration	Decision	Optimisation
Unbalanced Tree Search	✓		
Numerical Semigroups	✓		
Subgraph Isomorphism		✓	
Clique Search		✓	✓
Knapsack			✓
Travelling Salesperson			✓

Table 6.8: Summary of application search types.

change, i.e. the call simply changes from `Seq<NodeGen, Optimisation, ...>` to `Seq<NodeGen, Decision, ...>`. Without the skeletons a user would be required to write custom search and node processing functions for each application.

All applications can use the four general purpose coordinations (Sequential, Depth-Bounded, Stack-Stealing, and Budget). Switching to using a new search coordination is a simple one word change, i.e. from `Seq<NodeGen, Optimisation, ...>` to `StackStealing<NodeGen, Optimisation, ...>`. For Depth-Bounded and Budget an additional search parameter (spawn depth and budget) must also be provided. Without the skeletons parallelissm would need to be manually inserted into each search application, requiring a large engineering effort.

It easy to migrate existing applications to use YewPar. For example, given the existing the Numerical Semigroups application (Section 5.2.1.2), creating a NodeGenerator required only 15 lines of code (and 10 lines to define serialisation). That is, with 25 extra lines of code we get a search that can be parallelised using four different coordination methods.

6.11 Summary

This chapter evaluated the performance of the skeletons on seven applications and over 25 instances.

We have shown that the use of general-purpose skeletons does not significantly degrade (sequential) performance compared to hand-coded searches, showing a mean sequential slowdown of just over 6.1% (Section 6.3) over 21 instances.

One challenge of searches is the propagation of knowledge to all workers as it is found. In Section 6.4 we show that global knowledge management using a partitioned global address space and bounds broadcasting is appropriate, even for medium sized clusters (255 workers), due to the limited number, and spread out nature, of updates.

The performance of the Depth-Bounded skeletons is explored in Section 6.5. We show the difficulty in choosing a good value for d_{cutoff} and that choosing a poor value can result in

significant slowdown. In general Depth-Bounded works well for Maximum Clique, TSP and SIP, but struggles to parallelise Knapsack, Numerical Semigroups and some UTS instances. Maximum speedups of 122 on 120 workers are observed, as are absolute slowdowns in the case of Knapsack and UTS. Analysing the choice of workpool, i.e. deque versus depth-pool scheduling (Section 6.5.1), shows little difference in performance. Given that the depth-pool structure is designed to respect heuristic orderings, we advocate its usage as a more principled approach.

The performance of the Stack-Stealing skeletons is explored in Section 6.6. In general Stack-Stealing works well for most instances, successfully bringing running times down to a few minutes. Stack-Stealing does struggle to parallelise Knapsack in some cases, however, due to random work-stealing, there can be a large variance in results. We show that the chunking optimisation for Stack-Stealing is often not beneficial, and that, even though the chunk sizes themselves vary significantly across the instances, there is no relation between chunk size and overall runtime performance.

The performance of the Budget skeletons is explored in Section 6.7. The difficulty of choosing a good budget value is shown, with the surprising conclusion that $budget = 10^5$ works well, but not necessarily best, for most applications and instances. In the best case, Budget performs well for all applications, including Knapsack that Depth-Bounded and Stack-Stealing struggled to parallelise.

The skeletons are compared in Section 6.8. When run with 120 workers and the best configuration parameters, no single skeleton performs well in all cases. Depth-Bounded performs best for Maximum Clique and Travelling Salesperson, Stack-Stealing performs best for many SIP instances, and Budget performs well for Knapsack, Numerical Semigroups and UTS. Often we observe similar performance between Stack-Stealing and Budget. In general, if it is not possible to know the best parameter settings, then Stack-Stealing should be used as this provides the highest average performance across all instances, although it may not be optimal for a specific application.

Finally, we apply the skeletons to larger instances, including the search for spreads in $\mathrm{H}(4, 2^2)$ and $\mathrm{H}(4, 3^2)$, on 255 workers (Section 6.9). For Maximum Clique and Finite Geometry all skeletons perform well, successfully bringing running times down to 5 minutes maximally, while often achieving greater than 60% efficiency. Depth-Bounded struggles to perform well for Numerical Semigroups as has been observed throughout the evaluation.

Chapter 7

Replicable Branch and Bound Search

Branch and bound searches depend on strong search ordering heuristics (Section 2.1.4.1) to improve performance by guiding search towards fruitful areas of the search tree, allowing early termination (decision) or improved bounds to be found quickly (optimisation).

Parallel search necessarily deviates from the sequential search order, and hence the heuristic order, often in dynamic and unpredictable ways, e.g. due to random work-stealing. Disruptions in the search order can lead to performance anomalies (Section 2.3.2.1) that make it difficult to reason about the performance of parallel search. For example, we may observe highly variable runtimes or absolute slowdowns when introducing parallelism.

Difficulty in reasoning about search performance has implications for domains such as empirical algorithm design. Ideally we should be able to compare search algorithms, i.e. different Node Generators, on parallel architectures to allow larger instances to be considered. However, given high runtime variability, it is often difficult to separate algorithmic improvement from performance fluctuations introduced by parallelism. That is, is one search algorithm faster than the other, or was the performance improvement due to an anomaly causing an improved bound to be quickly found?

This chapter designs and evaluates specialised search skeletons, the *Ordered* skeletons, that guarantee the following replicable performance properties:

Sequential Lower Bound: Parallel runtime is never higher than the sequential (one worker) runtime.

Non-increasing Runtimes: Parallel runtime does not increase as the number of workers increases.

Repeatability: Parallel runtimes of repeated searches on the same parallel configuration have low variance.

Using MT^3 (Chapter 3) we show by example how search anomalies can occur and how, by carefully controlling parallel task orderings, the replicable properties are achieved (Section 7.2). A new search coordination method, designed to achieve these properties, is discussed in Section 7.4 and empirically evaluated to show that the replicable search properties are achieved in practice (Section 7.5).

The Ordered skeletons were previously implemented in the prototype Haskell framework described in Section 5.1.1 and [4].

7.1 Limits of Replicable Search

Replicable search only applies to branch and bound decision and optimisation searches. Enumeration searches do not suffer from anomalies as there is no information sharing between sub-trees[1].

The Ordered skeletons only provide **performance** replicability. Searches themselves may return different results, e.g. if there are two or more valid optimal solutions. This allows performance benefits from finding (different) improved solutions early, potentially leading to superlinear speedups (acceleration anomalies). For optimisation problems replicable solutions may be provided by returning all optimal solutions[2]. Such an approach cannot be applied to decision searches unless early termination is disabled.

We only guarantee the replicable performance properties with respect to search order effects, i.e. those that cause performance anomalies. Flexibility is required in the properties to allow for parallelism overheads. The **sequential lower bound** property is given relative to the one worker (parallel) runtime, rather than a strictly sequential search. This allows parallel overheads, e.g. spawning and scheduling tasks, to be accounted for. Likewise small slowdowns in the **non-increasing runtimes** property are acceptable as these are likely caused by parallel overheads of managing additional workers. Large slowdowns, likely to caused by search anomalies, are not.

We measure repeatability using relative standard deviation (RSD). RSD is defined as the (percentage) ratio of sample standard deviation to sample mean, i.e. $\frac{stdev(runtime)}{mean(runtime)} \times 100$. RSD represents how dispersed the samples are. For example, an RSD of 20% implies that we expect 68% of runtimes (a standard deviations worth) to fall within 20% of the sample mean.

While we may say process X is more repeatable than process Y, what may be considered a *good enough* value of RSD depends on the context it is being used in. In Appendix C we give one interpretation of repeatability, repeatability of scaling, that suggests an RSD of less than

[1] In practice the Ordered search coordination can be used to perform enumeration searches but there is no benefit over the other, often better-performing, skeletons.

[2] Alternatively we can use an injective objective function to provide an ordering on solutions.

17% may be appropriate to detect perfect scaling, and am RSD less than 10% is appropriate for instances that scale less well. This is only one interpretation of repeatability. Many are possible depending on circumstance, e.g. if we are considering the effect of algorithmic changes we may require tighter RSD bounds. We report only the value of repeatability in comparison to the other skeletons and leave the interpretation open.

7.2 Characterising Anomalies in MT^3

The conditions where anomalies occur, and the necessary steps to avoid detrimental/deceleration anomalies, are well studied in the literature [47, 48, 49, 50]. Proofs are given by analysing properties of search, in particular the node selection function (known as a heuristic function) and lower bound functions. While example trees are given to emphasise specific scenarios, specific search reductions are not.

This section extends the existing literature by showing how anomalies can occur using example parallel tree reductions in MT^3 (Chapter 3).

7.2.1 Anomalies

As discussed in Section 2.3.2.1, search order anomalies can occur when a parallel search performs a different amount of work than a sequential search for the same instance. If the parallel search performs less work than a sequential search then an acceleration anomaly may occur, allowing super-linear speedups. Inversely, if the parallel search performs significantly more overall work than a sequential search then detrimental anomalies *may* appear.

Ideally we wish to avoid detrimental anomalies while allowing the possibility of acceleration anomalies, i.e. parallelism should make things faster but never slower.

To show how anomalies occur we use the (synthetic) search tree of Figure 7.1. As the exact node structure in unimportant we identify nodes using their position labels. We further assume that the task set is constructed statically ahead of time and consists of all tasks at depth 1 (the sub-trees rooted at 00, 01, 02, 03, and 04).

Like many search trees, the heuristics cause nodes with good bounds to be clustered to the left of the tree. In this case the node with the optimal solution (0100) is not in the leftmost branch. Instead it is in the branch with one discrepancy showing the heuristic is not perfect. Parallelism counteracts non-perfect heuristics by speculatively exploring additional branches.

To analyse anomalous behaviour we add timing information to MT^3 by assuming that in a single timestep we perform one round-robin schedule of all threads, e.g. for two threads we can perform up to two reduction rules (Section 3.3) in a timestep. We assume that each rule

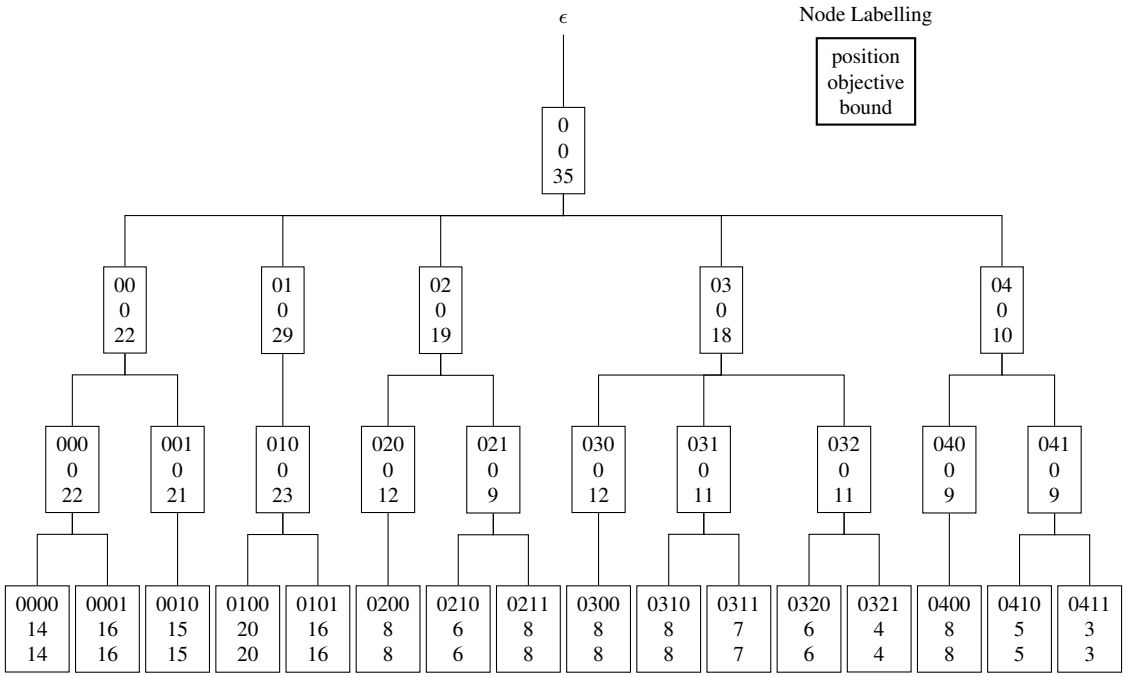


Figure 7.1: Synthetic search tree to show how anomalies affect search.

has uniform cost. Within a particular round-robin schedule we work from thread 1 to thread n and any global state updates made by thread $i-1$ are instantly available to thread i.

For brevity we write $S_{00\backslash 000}$ to mean $(S_{00} \setminus S_{000}) \cup \{000\}$, i.e. the tree rooted at 00 with the sub-tree rooted at 000 removed, still containing 000. We also let $S_0{:}T$ correspond to the non-empty task list T where S_0 is the next element to be scheduled.

The one worker reduction of the search tree in Figure 7.1 is given in Figure 7.2. Due to the good heuristic ordering, the one worker search requires only 29 time steps to complete. By finding the optimal solution early in the second left-most branch the thread is able to avoid exploring the sub-trees rooted at 02, 03 and 04 completely.

The two worker reduction of the same search tree is given in Figure 7.3. Parallelism significantly improves the search by finding the optimal solution much earlier than the one worker search, $N = 4$ instead of $N = 16$. This shows a major benefit of parallel search, by speculatively exploring the sub-tree S_{01} we account for the non-perfect heuristic function. Importantly, the knowledge is flowing right-to-left in the tree. While, in this case, early knowledge only helps avoid an additional optimise step ($N = 7$ in Figure 7.2), in practice larger parts of the search space is often eliminated. By completing in 14 steps instead of 29 steps we see that parallelism gives us a better than linear speedup and an acceleration anomaly has occurred.

In practice parallel tree search often contains an element of randomness e.g. due to (random) work-stealing. What if, due to this randomness, the workqueue ended up in the worst

Timestep	Rule	State	Obj	Bound	Best
0		$\langle\{\epsilon\}_o, [S_{00}, S_{01}, S_{02}, S_{03}, S_{,04}\,], \bot\rangle$	$\langle\bot\rangle$	$\langle\bot\rangle$	0
1	(schedule$_1$)	$\langle\{\epsilon\}_o, S_{01}{:}T, \langle S_{00}, 00\rangle\rangle$	$\langle 0\rangle$	$\langle 35\rangle$	0
2	(advance$_1$)	$\langle\{\epsilon\}_o, S_{01}{:}T, \langle S_{00}, 000\rangle\rangle$	$\langle 0\rangle$	$\langle 22\rangle$	0
3	(advance$_1$)	$\langle\{\epsilon\}_o, S_{01}{:}T, \langle S_{00}, 0000\rangle\rangle$	$\langle 14\rangle$	$\langle 14\rangle$	0
4	(optimise$_1$)	$\langle\{0000\}_o, S_{01}{:}T, \langle S_{00}, 0000\rangle\rangle$	$\langle 14\rangle$	$\langle 14\rangle$	14
5	(prune-opt$_1$)	$\langle\{0000\}_o, S_{01}{:}T, \langle S_{00\setminus 0000}, 0000\rangle\rangle$	$\langle 14\rangle$	$\langle 14\rangle$	14
6	(advance$_1$)	$\langle\{0000\}_o, S_{01}{:}T, \langle S_{00\setminus 0000}, 0001\rangle\rangle$	$\langle 16\rangle$	$\langle 16\rangle$	14
7	(optimise$_1$)	$\langle\{0001\}_o, S_{01}{:}T, \langle S_{00\setminus 0000}, 0001\rangle\rangle$	$\langle 16\rangle$	$\langle 16\rangle$	16
8	(prune-opt$_1$)	$\langle\{0001\}_o, S_{01}{:}T, \langle S_{00\setminus 0000\setminus 0001}, 0001\rangle\rangle$	$\langle 16\rangle$	$\langle 16\rangle$	16
9	(advance$_1$)	$\langle\{0001\}_o, S_{01}{:}T, \langle S_{00\setminus 0000\setminus 0001}, 001\rangle\rangle$	$\langle 0\rangle$	$\langle 21\rangle$	16
10	(advance$_1$)	$\langle\{0001\}_o, S_{01}{:}T, \langle S_{00\setminus 0000\setminus 0001}, 0010\rangle\rangle$	$\langle 15\rangle$	$\langle 15\rangle$	16
11	(prune-opt$_1$)	$\langle\{0001\}_o, S_{01}{:}T, \langle S_{00\setminus 0000\setminus 0001\setminus 0010}, 0010\rangle\rangle$	$\langle 15\rangle$	$\langle 15\rangle$	16
12	(terminate$_1$)	$\langle\{0001\}_o, S_{01}{:}T, \bot\rangle$	$\langle\bot\rangle$	$\langle\bot\rangle$	16
13	(schedule$_1$)	$\langle\{0001\}_o, S_{02}{:}T, \langle S_{01}, 01\rangle\rangle$	$\langle 0\rangle$	$\langle 29\rangle$	16
14	(advance$_1$)	$\langle\{0001\}_o, S_{02}{:}T, \langle S_{01}, 010\rangle\rangle$	$\langle 0\rangle$	$\langle 23\rangle$	16
15	(advance$_1$)	$\langle\{0001\}_o, S_{02}{:}T, \langle S_{01}, 0100\rangle\rangle$	$\langle 20\rangle$	$\langle 20\rangle$	16
16	(optimise$_1$)	$\langle\{0100\}_o, S_{02}{:}T, \langle S_{01}, 0100\rangle\rangle$	$\langle 20\rangle$	$\langle 20\rangle$	20
17	(prune-opt$_1$)	$\langle\{0100\}_o, S_{02}{:}T, \langle S_{01\setminus 0100}, 0100\rangle\rangle$	$\langle 20\rangle$	$\langle 20\rangle$	20
18	(advance$_1$)	$\langle\{0100\}_o, S_{02}{:}T, \langle S_{01\setminus 0100}, 0101\rangle\rangle$	$\langle 16\rangle$	$\langle 16\rangle$	20
19	(prune-opt$_1$)	$\langle\{0100\}_o, S_{02}{:}T, \langle S_{01\setminus 0100\setminus 0101}, 0101\rangle\rangle$	$\langle 16\rangle$	$\langle 16\rangle$	20
20	(terminate$_1$)	$\langle\{0100\}_o, S_{02}{:}T, \bot\rangle$	$\langle\bot\rangle$	$\langle\bot\rangle$	20
21	(schedule$_1$)	$\langle\{0100\}_o, S_{03}{:}T, \langle S_{02}, 02\rangle\rangle$	$\langle 0\rangle$	$\langle 19\rangle$	20
22	(prune-opt$_1$)	$\langle\{0100\}_o, S_{03}{:}T, \langle S_{02\setminus 02}, 02\rangle\rangle$	$\langle 0\rangle$	$\langle 19\rangle$	20
23	(terminate$_1$)	$\langle\{0100\}_o, S_{03}{:}T, \bot\rangle$	$\langle\bot\rangle$	$\langle\bot\rangle$	20
24	(schedule$_1$)	$\langle\{0100\}_o, S_{04}{:}[\,], \langle S_{03}, 03\rangle\rangle$	$\langle 0\rangle$	$\langle 18\rangle$	20
25	(prune-opt$_1$)	$\langle\{0100\}_o, S_{04}{:}[\,], \langle S_{03\setminus 03}, 03\rangle\rangle$	$\langle 0\rangle$	$\langle 18\rangle$	20
26	(terminate$_1$)	$\langle\{0100\}_o, S_{04}{:}[\,], \bot\rangle$	$\langle\bot\rangle$	$\langle\bot\rangle$	20
27	(schedule$_1$)	$\langle\{0100\}_o, [\,], \langle S_{04}, 04\rangle\rangle$	$\langle 0\rangle$	$\langle 10\rangle$	20
28	(prune-opt$_1$)	$\langle\{0100\}_o, [\,], \langle S_{04\setminus 04}, 04\rangle\rangle$	$\langle 0\rangle$	$\langle 10\rangle$	20
29	(terminate$_1$)	$\langle\{0100\}_o, [\,], \bot\rangle$	$\langle\bot\rangle$	$\langle\bot\rangle$	20

Figure 7.2: One worker reduction of the search tree in Figure 7.1. **Obj** and **Bound** show the current objective value and bound for the node currently being viewed by each thread. **Best** is the current incumbent objective value.

Timestep	Rule	State	Obj	Bound	Best
0		$\langle \{\epsilon\}_o, [S_{00}, S_{01}, S_{02}, S_{03}, S_{04}], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	0
1	(schedule$_1$)	$\langle \{\epsilon\}_o, S_{01}{:}T, \langle S_{00}, 00 \rangle, \bot \rangle$	$\langle 0, \bot \rangle$	$\langle 22, \bot \rangle$	0
1	(schedule$_2$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{00}, 00 \rangle, \langle S_{01}, 01 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 22, 29 \rangle$	0
2	(advance$_1$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{00}, 000 \rangle, \langle S_{01}, 01 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 22, 29 \rangle$	0
2	(advance$_2$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{00}, 000 \rangle, \langle S_{01}, 010 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 22, 23 \rangle$	0
3	(advance$_1$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{00}, 0000 \rangle, \langle S_{01}, 010 \rangle \rangle$	$\langle 14, 0 \rangle$	$\langle 14, 23 \rangle$	0
3	(advance$_2$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{00}, 0000 \rangle, \langle S_{01}, 0100 \rangle \rangle$	$\langle 14, 20 \rangle$	$\langle 14, 20 \rangle$	0
4	(optimise$_1$)	$\langle \{0000\}_o, S_{02}{:}T, \langle S_{00}, 0000 \rangle, \langle S_{01}, 0100 \rangle \rangle$	$\langle 14, 20 \rangle$	$\langle 14, 20 \rangle$	14
4	(optimise$_2$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00}, 0000 \rangle, \langle S_{01}, 0100 \rangle \rangle$	$\langle 14, 20 \rangle$	$\langle 14, 20 \rangle$	20
5	(prune-opt$_1$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000}, 0000 \rangle, \langle S_{01}, 0100 \rangle \rangle$	$\langle 14, 20 \rangle$	$\langle 14, 20 \rangle$	20
5	(prune-opt$_2$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000}, 0000 \rangle, \langle S_{01 \setminus 0100}, 0100 \rangle \rangle$	$\langle 14, 20 \rangle$	$\langle 14, 20 \rangle$	20
6	(advance$_1$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000}, 0001 \rangle, \langle S_{01 \setminus 0100}, 0100 \rangle \rangle$	$\langle 16, 20 \rangle$	$\langle 16, 20 \rangle$	20
6	(advance$_2$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000}, 0001 \rangle, \langle S_{01 \setminus 0100}, 0101 \rangle \rangle$	$\langle 16, 16 \rangle$	$\langle 16, 16 \rangle$	20
7	(prune-opt$_1$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000 \setminus 0001}, 0001 \rangle, \langle S_{01 \setminus 0100}, 0101 \rangle \rangle$	$\langle 16, 16 \rangle$	$\langle 16, 16 \rangle$	20
7	(prune-opt$_2$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000 \setminus 0001}, 0001 \rangle, \langle S_{01 \setminus 0100 \setminus 0101}, 0101 \rangle \rangle$	$\langle 16, 16 \rangle$	$\langle 16, 16 \rangle$	20
8	(advance$_1$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000 \setminus 0001}, 001 \rangle, \langle S_{01 \setminus 0100 \setminus 0101}, 0101 \rangle \rangle$	$\langle 0, 16 \rangle$	$\langle 21, 16 \rangle$	20
8	(terminate$_1$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000 \setminus 0001}, 001 \rangle, \bot \rangle$	$\langle 0, \bot \rangle$	$\langle 21, \bot \rangle$	20
9	(advance$_1$)	$\langle \{0100\}_o, S_{02}{:}T, \langle S_{00 \setminus 0000 \setminus 0001}, 0010 \rangle, \bot \rangle$	$\langle 15, \bot \rangle$	$\langle 15, \bot \rangle$	20
9	(schedule$_2$)	$\langle \{0100\}_o, S_{03}{:}T, \langle S_{00 \setminus 0000 \setminus 0001}, 0010 \rangle, \langle S_{02}, 02 \rangle \rangle$	$\langle 15, 0 \rangle$	$\langle 15, 19 \rangle$	20
10	(prune-opt$_1$)	$\langle \{0100\}_o, S_{03}{:}T, \langle S_{00 \setminus 0000 \setminus 0001 \setminus 0010}, 0010 \rangle, \langle S_{02}, 02 \rangle \rangle$	$\langle 15, 0 \rangle$	$\langle 15, 19 \rangle$	20
10	(prune-opt$_2$)	$\langle \{0100\}_o, S_{03}{:}T, \langle S_{00 \setminus 0000 \setminus 0001 \setminus 0010}, 0010 \rangle, \langle S_{02 \setminus 02}, 02 \rangle \rangle$	$\langle 15, 0 \rangle$	$\langle 15, 19 \rangle$	20
11	(terminate$_1$)	$\langle \{0100\}_o, S_{03}{:}T, \bot, \langle S_{02 \setminus 02}, 02 \rangle \rangle$	$\langle \bot, 0 \rangle$	$\langle \bot, 19 \rangle$	20
11	(terminate$_2$)	$\langle \{0100\}_o, S_{03}{:}T, \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	20
12	(schedule$_1$)	$\langle \{0100\}_o, S_{04}{:}[\,], \langle S_{03}, 03 \rangle, \bot \rangle$	$\langle 0, \bot \rangle$	$\langle 18, \bot \rangle$	20
12	(schedule$_2$)	$\langle \{0100\}_o, [\,], \langle S_{03}, 03 \rangle, \langle S_{04}, 04 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 18, 10 \rangle$	20
13	(prune-opt$_1$)	$\langle \{0100\}_o, [\,], \langle S_{03 \setminus 03}, 03 \rangle, \langle S_{04}, 04 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 18, 10 \rangle$	20
13	(prune-opt$_2$)	$\langle \{0100\}_o, [\,], \langle S_{03 \setminus 03}, 03 \rangle, \langle S_{04 \setminus 04}, 04 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 18, 10 \rangle$	20
14	(terminate$_1$)	$\langle \{0100\}_o, [\,], \bot, \langle S_{04 \setminus 04}, 04 \rangle \rangle$	$\langle \bot, 0 \rangle$	$\langle \bot, 10 \rangle$	20
14	(terminate$_2$)	$\langle \{0100\}_o, [\,], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	20

Figure 7.3: Two worker reduction of the search tree in Figure 7.1, with initial tasks following the heuristic order. **Obj** and **Bound** show the current objective value and bound for the node currently being viewed by each thread. **Best** is the current incumbent objective value.

heuristic order[3]: $[S_{04}, S_{03}, S_{02}, S_{01}, S_{00}]$? The two worker reduction in this situation is given in Figure 7.4.

This is an example of a slowdown anomaly as the two workers require 30 timesteps to complete compared to 29 with one worker. Due to the poor ordering, the search spends most of the time in the right half of the tree where, as search never finds a good bound, limited pruning is possible causing the additional work.

The determining factor of this slowdown is *when* the optimal solution is found. As it is found later than the one worker case, step 19 instead of step 16, more work is performed on sub-trees that would have been pruned in the one worker case.

Figure 7.5 shows the same two worker parallel reduction as Figure 7.4, however this time we update the incumbent to the optimal value at $N = 16$ to match the one worker case. Armed with the knowledge about the optimal solution, the search completes in 28 steps showing a small improvement over the one worker search. Importantly it is never slower than the one worker case.

If however, as in Figure 7.6, the incumbent is not updated until $N = 17$, i.e. after the one worker case, the slowdown anomaly remains. This example suggests that, in order to perform no worse than the one worker case, we require the optimal solution in the two worker search to be found before, or at the same time as, the one worker search. This is formalised in the following section.

7.3 Achieving Replicable Search

An important feature of the anomaly avoidance literature [47, 48, 49, 50] is the choice of heuristic function, i.e. the function that picks the next node to expand, for example depth-first or best-first. To avoid slowdowns over sequential we must ensure that, at any point in the execution, at least one node that would have been expanded in the sequential search[4] is expanded in the parallel search. This can be achieved by having a consistent ordering on the heuristic function for both sequential and parallel runs. Li and Wah [48] obtain such an ordering in a similar manner to MT^3: by labelling tree nodes with a unique path and using this to tie break between similar nodes (e.g. nodes of the same value in a best-first search).

The ordering on nodes implies an ordering on the time that knowledge is learned. As we deal specifically with depth-first search, instead of focusing on the heuristic function as in the previous work, we instead focus on knowledge. The conditions required to avoid anomalies remain the same as previous work.

[3]Given perfect information, the worst order is $[S_{04}, S_{03}, S_{02}, S_{00}, S_{01}]$ as the solution is in S_{01}. To emulate a real search that does not have solution information we use the worst *heuristic* order, i.e. reverse child order.

[4]So called primary nodes [47, 50], E-nodes [49] or basic nodes [48].

Timestep	Rule	State	Obj	Bound	Best
0		$\langle \{\epsilon\}_o, [S_{04}, S_{03}, S_{02}, S_{01}, S_{00}], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	0
1	(schedule$_1$)	$\langle \{\epsilon\}_o, S_{03}{:}T, \langle S_{04}, 04 \rangle, \bot \rangle$	$\langle 0, \bot \rangle$	$\langle 10, \bot \rangle$	0
1	(schedule$_2$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{04}, 04 \rangle, \langle S_{03}, 03 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 10, 18 \rangle$	0
2	(advance$_1$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{04}, 040 \rangle, \langle S_{03}, 03 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 9, 18 \rangle$	0
2	(advance$_2$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{04}, 040 \rangle, \langle S_{03}, 030 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 9, 12 \rangle$	0
3	(advance$_1$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{04}, 0400 \rangle, \langle S_{03}, 030 \rangle \rangle$	$\langle 8, 0 \rangle$	$\langle 8, 12 \rangle$	0
3	(advance$_2$)	$\langle \{\epsilon\}_o, S_{02}{:}T, \langle S_{04}, 0400 \rangle, \langle S_{03}, 0300 \rangle \rangle$	$\langle 8, 8 \rangle$	$\langle 8, 8 \rangle$	0
4	(optimise$_1$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04}, 0400 \rangle, \langle S_{03}, 0300 \rangle \rangle$	$\langle 8, 8 \rangle$	$\langle 8, 8 \rangle$	8
4	(prune-opt$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04}, 0400 \rangle, \langle S_{03 \setminus 0300}, 0300 \rangle \rangle$	$\langle 8, 8 \rangle$	$\langle 8, 8 \rangle$	8
5	(prune-opt$_1$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400}, 0400 \rangle, \langle S_{03 \setminus 0300}, 0300 \rangle \rangle$	$\langle 8, 8 \rangle$	$\langle 8, 8 \rangle$	8
5	(advance$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400}, 0400 \rangle, \langle S_{03 \setminus 0300}, 031 \rangle \rangle$	$\langle 8, 0 \rangle$	$\langle 8, 11 \rangle$	8
6	(advance$_1$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400}, 041 \rangle, \langle S_{03 \setminus 0300}, 031 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 9, 11 \rangle$	8
6	(advance$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400}, 041 \rangle, \langle S_{03 \setminus 0300}, 0310 \rangle \rangle$	$\langle 0, 8 \rangle$	$\langle 9, 8 \rangle$	8
7	(advance$_1$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400}, 0410 \rangle, \langle S_{03 \setminus 0300}, 0310 \rangle \rangle$	$\langle 5, 8 \rangle$	$\langle 5, 8 \rangle$	8
7	(prune-opt$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400}, 0410 \rangle, \langle S_{03 \setminus 0300 \setminus 0310}, 0310 \rangle \rangle$	$\langle 5, 8 \rangle$	$\langle 5, 8 \rangle$	8
8	(prune-opt$_1$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400 \setminus 0410}, 0410 \rangle, \langle S_{03 \setminus 0300 \setminus 0310}, 0310 \rangle \rangle$	$\langle 5, 8 \rangle$	$\langle 5, 8 \rangle$	8
8	(advance$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400 \setminus 0410}, 0410 \rangle, \langle S_{03 \setminus 0300 \setminus 0310}, 0311 \rangle \rangle$	$\langle 5, 7 \rangle$	$\langle 5, 7 \rangle$	8
9	(advance$_1$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400 \setminus 0410}, 0411 \rangle, \langle S_{03 \setminus 0300 \setminus 0310}, 0311 \rangle \rangle$	$\langle 5, 7 \rangle$	$\langle 5, 7 \rangle$	8
9	(prune-opt$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400 \setminus 0410}, 0411 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311}, 0311 \rangle \rangle$	$\langle 5, 7 \rangle$	$\langle 5, 7 \rangle$	8
10	(prune-opt$_1$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400 \setminus 0410 \setminus 0411}, 0411 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311}, 0311 \rangle \rangle$	$\langle 5, 7 \rangle$	$\langle 5, 7 \rangle$	8
10	(advance$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \langle S_{04 \setminus 0400 \setminus 0410 \setminus 0411}, 0411 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311}, 032 \rangle \rangle$	$\langle 5, 0 \rangle$	$\langle 5, 11 \rangle$	8
11	(terminate$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \bot, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311}, 032 \rangle \rangle$	$\langle \bot, 0 \rangle$	$\langle \bot, 11 \rangle$	8
11	(advance$_2$)	$\langle \{0400\}_o, S_{02}{:}T, \bot, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311}, 0320 \rangle \rangle$	$\langle \bot, 6 \rangle$	$\langle \bot, 6 \rangle$	8
12	(schedule$_1$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02}, 02 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311}, 0320 \rangle \rangle$	$\langle 0, 6 \rangle$	$\langle 19, 6 \rangle$	8
12	(prune-opt$_2$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02}, 02 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311 \setminus 0320}, 0320 \rangle \rangle$	$\langle 0, 6 \rangle$	$\langle 9, 6 \rangle$	8
13	(advance$_1$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02}, 020 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311 \setminus 0320}, 0320 \rangle \rangle$	$\langle 0, 6 \rangle$	$\langle 12, 6 \rangle$	8
13	(advance$_2$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02}, 020 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311 \setminus 0320}, 0321 \rangle \rangle$	$\langle 0, 4 \rangle$	$\langle 12, 4 \rangle$	8
14	(advance$_1$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02}, 0200 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311 \setminus 0320}, 0321 \rangle \rangle$	$\langle 8, 4 \rangle$	$\langle 8, 4 \rangle$	8
14	(prune-opt$_2$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02}, 0200 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311 \setminus 0320 \setminus 0321}, 0321 \rangle \rangle$	$\langle 8, 4 \rangle$	$\langle 8, 4 \rangle$	8
15	(prune-opt$_1$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02 \setminus 0200}, 0200 \rangle, \langle S_{03 \setminus 0300 \setminus 0310 \setminus 0311 \setminus 0320 \setminus 0321}, 0321 \rangle \rangle$	$\langle 8, 4 \rangle$	$\langle 8, 4 \rangle$	8
15	(terminate$_2$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02 \setminus 0200}, 0200 \rangle, \bot \rangle$	$\langle 8, \bot \rangle$	$\langle 8, \bot \rangle$	8
16	(advance$_1$)	$\langle \{0400\}_o, S_{01}{:}T, \langle S_{02 \setminus 0200}, 021 \rangle, \bot \rangle$	$\langle 0, \bot \rangle$	$\langle 9, \bot \rangle$	8
16	(schedule$_2$)	$\langle \{0400\}_o, S_{00}{:}[], \langle S_{02 \setminus 0200}, 021 \rangle, \langle S_{01}, 01 \rangle \rangle$	$\langle 0, 0 \rangle$	$\langle 9, 29 \rangle$	8
17	(advance$_1$)	$\langle \{0400\}_o, S_{00}{:}[], \langle S_{02 \setminus 0200}, 0210 \rangle, \langle S_{01}, 01 \rangle \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 29 \rangle$	8
17	(advance$_2$)	$\langle \{0400\}_o, S_{00}{:}[], \langle S_{02 \setminus 0200}, 0210 \rangle, \langle S_{01}, 010 \rangle \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 23 \rangle$	8
18	(prune-opt$_1$)	$\langle \{0400\}_o, S_{00}{:}[], \langle S_{02 \setminus 0200 \setminus 0210}, 0210 \rangle, \langle S_{01}, 010 \rangle \rangle$	$\langle 6, 0 \rangle$	$\langle 6, 23 \rangle$	8
18	(advance$_2$)	$\langle \{0400\}_o, S_{00}{:}[], \langle S_{02 \setminus 0200 \setminus 0210}, 0210 \rangle, \langle S_{01}, 0100 \rangle \rangle$	$\langle 6, 20 \rangle$	$\langle 6, 20 \rangle$	8
19	(prune-opt$_1$)	$\langle \{0400\}_o, S_{00}{:}[], \langle S_{02 \setminus 0200 \setminus 0210 \setminus 0211}, 0211 \rangle, \langle S_{01}, 0100 \rangle \rangle$	$\langle 6, 8 \rangle$	$\langle 6, 8 \rangle$	8
19	(optimise$_2$)	$\langle \{0100\}_o, S_{00}{:}[], \langle S_{02 \setminus 0200 \setminus 0210 \setminus 0211}, 0211 \rangle, \langle S_{01}, 0100 \rangle \rangle$	$\langle 6, 8 \rangle$	$\langle 6, 8 \rangle$	20
20	(terminate$_1$)	$\langle \{0100\}_o, S_{00}{:}[], \bot, \langle S_{01}, 0100 \rangle \rangle$	$\langle \bot, 8 \rangle$	$\langle \bot, 8 \rangle$	20
20	(prune-opt$_2$)	$\langle \{0100\}_o, S_{00}{:}[], \bot, \langle S_{01 \setminus 0100}, 0100 \rangle \rangle$	$\langle \bot, 8 \rangle$	$\langle \bot, 8 \rangle$	20
21	(schedule$_1$)	$\langle \{0100\}_o, [], \langle S_{00}, 00 \rangle, \langle S_{01 \setminus 0100}, 0100 \rangle \rangle$	$\langle 0, 8 \rangle$	$\langle 22, 8 \rangle$	20
21	(advance$_2$)	$\langle \{0100\}_o, [], \langle S_{00}, 00 \rangle, \langle S_{01 \setminus 0100}, 0101 \rangle \rangle$	$\langle 0, 8 \rangle$	$\langle 22, 8 \rangle$	20
22	(advance$_1$)	$\langle \{0100\}_o, [], \langle S_{00}, 000 \rangle, \langle S_{01 \setminus 0100}, 0101 \rangle \rangle$	$\langle 0, 8 \rangle$	$\langle 22, 8 \rangle$	20
22	(prune-opt$_2$)	$\langle \{0100\}_o, [], \langle S_{00}, 000 \rangle, \langle S_{01 \setminus 0100 \setminus 0101}, 0101 \rangle \rangle$	$\langle 0, 8 \rangle$	$\langle 22, 8 \rangle$	20
23	(advance$_1$)	$\langle \{0100\}_o, [], \langle S_{00}, 0000 \rangle, \langle S_{01 \setminus 0100 \setminus 0101}, 0101 \rangle \rangle$	$\langle 14, 8 \rangle$	$\langle 14, 8 \rangle$	20
23	(terminate$_2$)	$\langle \{0100\}_o, [], \langle S_{00}, 0000 \rangle, \bot \rangle$	$\langle 14, \bot \rangle$	$\langle 14, \bot \rangle$	20
24	(prune-opt$_1$)	$\langle \{0100\}_o, [], \langle S_{00 \setminus 0000}, 0000 \rangle, \bot \rangle$	$\langle 14, \bot \rangle$	$\langle 14, \bot \rangle$	20
25	(advance$_1$)	$\langle \{0100\}_o, [], \langle S_{00 \setminus 0000}, 0001 \rangle, \bot \rangle$	$\langle 16, \bot \rangle$	$\langle 16, \bot \rangle$	20
26	(prune-opt$_1$)	$\langle \{0100\}_o, [], \langle S_{00 \setminus 0000 \setminus 0001}, 0001 \rangle, \bot \rangle$	$\langle 16, \bot \rangle$	$\langle 16, \bot \rangle$	20
27	(advance$_1$)	$\langle \{0100\}_o, [], \langle S_{00 \setminus 0000 \setminus 0001}, 001 \rangle, \bot \rangle$	$\langle 0, \bot \rangle$	$\langle 21, \bot \rangle$	20
28	(advance$_1$)	$\langle \{0100\}_o, [], \langle S_{00 \setminus 0000 \setminus 0001}, 0010 \rangle, \bot \rangle$	$\langle 15, \bot \rangle$	$\langle 21, \bot \rangle$	20
29	(prune-opt$_1$)	$\langle \{0100\}_o, [], \langle S_{00 \setminus 0000 \setminus 0001 \setminus 0010}, 0010 \rangle, \bot \rangle$	$\langle 15, \bot \rangle$	$\langle 21, \bot \rangle$	20
30	(terminate$_1$)	$\langle \{0100\}_o, [], \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	20

Figure 7.4: Two worker reduction of the search tree in Figure 7.1, with initial tasks going against the heuristic order. **Obj** and **Bound** show the current objective value and bound for the node currently being viewed by each thread. **Best** is the current incumbent objective value.

Timestep	Rule	State	Obj	Bound	Best
0		$\langle\{\epsilon\}_o, [S_{04}, S_{03}, S_{02}, S_{01}, S_{00}], \bot, \bot\rangle$	$\langle\bot, \bot\rangle$	$\langle\bot, \bot\rangle$	0
1	(schedule$_1$)	$\langle\{\epsilon\}_o, S_{03}{:}T, \langle S_{04}, 04\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 10, \bot\rangle$	0
1	(schedule$_2$)	$\langle\{\epsilon\}_o, S_{02}{:}T, \langle S_{04}, 04\rangle, \langle S_{03}, 03\rangle\rangle$	$\langle 0, 0\rangle$	$\langle 10, 18\rangle$	0
...	...	...	...	...	...
16	(advance$_1$)	$\langle\{0400\}_o, S_{01}{:}T, \langle S_{02\setminus 0200}, 021\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 9, \bot\rangle$	8
16	(schedule$_2$)	$\langle\{0400\}_o, S_{00}{:}[], \langle S_{02\setminus 0200}, 021\rangle, \langle S_{01}, 01\rangle\rangle$	$\langle 0, 0\rangle$	$\langle 9, 29\rangle$	8
16	(**Incumbent Update**)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200}, 021\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 9, \bot\rangle$	20
17	(prune-opt$_1$)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200\setminus 021}, 021\rangle, \langle S_{01}, 01\rangle\rangle$	$\langle 0, 0\rangle$	$\langle 9, 29\rangle$	20
17	(advance$_2$)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200\setminus 021}, 021\rangle, \langle S_{01}, 010\rangle\rangle$	$\langle 0, 0\rangle$	$\langle 9, 23\rangle$	20
18	(terminate$_1$)	$\langle\{0100\}_o, S_{00}{:}[], \bot, \langle S_{01}, 010\rangle\rangle$	$\langle\bot, 0\rangle$	$\langle\bot, 23\rangle$	20
18	(advance$_2$)	$\langle\{0100\}_o, S_{00}{:}[], \bot, \langle S_{01}, 0100\rangle\rangle$	$\langle\bot, 20\rangle$	$\langle\bot, 20\rangle$	20
19	(schedule$_1$)	$\langle\{0100\}_o, [], \langle S_{00}, 00\rangle, \langle S_{01}, 0100\rangle\rangle$	$\langle 0, 20\rangle$	$\langle 22, 20\rangle$	20
19	(prune-opt$_2$)	$\langle\{0100\}_o, [], \langle S_{00}, 00\rangle, \langle S_{01\setminus 0100}, 0100\rangle\rangle$	$\langle 0, 20\rangle$	$\langle 22, 20\rangle$	20
20	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00}, 000\rangle, \langle S_{01\setminus 0100}, 0100\rangle\rangle$	$\langle 0, 20\rangle$	$\langle 22, 20\rangle$	20
20	(advance$_2$)	$\langle\{0100\}_o, [], \langle S_{00}, 000\rangle, \langle S_{01\setminus 0100}, 0101\rangle\rangle$	$\langle 0, 16\rangle$	$\langle 22, 16\rangle$	20
21	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00}, 0000\rangle, \langle S_{01\setminus 0100}, 0101\rangle\rangle$	$\langle 14, 16\rangle$	$\langle 14, 16\rangle$	20
21	(prune-opt$_2$)	$\langle\{0100\}_o, [], \langle S_{00}, 0000\rangle, \langle S_{01\setminus 0100\setminus 0101}, 0101\rangle\rangle$	$\langle 14, 16\rangle$	$\langle 14, 16\rangle$	20
22	(prune-opt$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000}, 0000\rangle, \langle S_{01\setminus 0100\setminus 0101}, 0101\rangle\rangle$	$\langle 14, 16\rangle$	$\langle 14, 16\rangle$	20
22	(terminate$_2$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000}, 0000\rangle, \bot\rangle$	$\langle 14, \bot\rangle$	$\langle 14, \bot\rangle$	20
23	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000}, 0001\rangle, \bot\rangle$	$\langle 16, \bot\rangle$	$\langle 16, \bot\rangle$	20
24	(prune-opt$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000\setminus 0001}, 0001\rangle, \bot\rangle$	$\langle 16, \bot\rangle$	$\langle 16, \bot\rangle$	20
25	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000\setminus 0001}, 001\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 21, \bot\rangle$	20
26	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000\setminus 0001}, 0010\rangle, \bot\rangle$	$\langle 15, \bot\rangle$	$\langle 15, \bot\rangle$	20
27	(prune-opt$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000\setminus 0001\setminus 0010}, 0010\rangle, \bot\rangle$	$\langle 15, \bot\rangle$	$\langle 15, \bot\rangle$	20
28	(terminate$_1$)	$\langle\{0100\}_o, [], \bot, \bot\rangle$	$\langle\bot, \bot\rangle$	$\langle\bot, \bot\rangle$	20

Figure 7.5: Two worker reduction of the search tree in Figure 7.1, with initial tasks going against the heuristic order. An additional update is made at timestep 16, as in the one worker reduction. ... represents the reduction from Figure 7.4. **Obj** and **Bound** show the current objective value and bound for the node currently being viewed by each thread. **Best** is the current incumbent objective value.

State Orderings To discuss the current knowledge of a search we define the following ordering on states, σ, such that:

$\{\}_d \sqsubset \{v\}_d$ That is, a state that contains a target node is stronger than one that does not.

$\{v\}_o \sqsubset \{u\}_o$ If $obj(v) < obj(u)$[5].

By ordering both decision and optimisation states, we show that the conditions for replicable search, that have largely been studied for optimisation problems only, also apply to decision searches.

Sequential Lower Bound The sequential lower bound requires no parallel search to be slower than the one worker search.

As we have shown (Section 7.2), for this to be the case we require an optimal/target node to be found in parallel before, or at the same time, it is found using a single worker. This

[5]Assuming a maximisation problem. Minimisation problems can be handled by having $\{v\}_o \sqsubset \{u\}_o$ if $obj(v) > obj(u)$.

Timestep	Rule	State	Obj	Bound	Best
0		$\langle\{\epsilon\}_o, [S_{04}, S_{03}, S_{02}, S_{01}, S_{00}], \bot, \bot\rangle$	$\langle\bot, \bot\rangle$	$\langle\bot, \bot\rangle$	0
1	(schedule$_1$)	$\langle\{\epsilon\}_o, S_{03}{:}T, \langle S_{04}, 04\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 10, \bot\rangle$	0
1	(schedule$_2$)	$\langle\{\epsilon\}_o, S_{02}{:}T, \langle S_{04}, 04\rangle, \langle S_{03}, 03\rangle\rangle$	$\langle 0, 0\rangle$	$\langle 10, 18\rangle$	0
...	...	...	...	...	...
16	(advance$_1$)	$\langle\{0400\}_o, S_{01}{:}T, \langle S_{02\setminus 0200}, 021\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 9, \bot\rangle$	8
16	(schedule$_2$)	$\langle\{0400\}_o, S_{00}{:}[], \langle S_{02\setminus 0200}, 021\rangle, \langle S_{01}, 01\rangle\rangle$	$\langle 0, 0\rangle$	$\langle 9, 29\rangle$	8
17	(advance$_1$)	$\langle\{0400\}_o, S_{00}{:}[], \langle S_{02\setminus 0200}, 0210\rangle, \langle S_{01}, 01\rangle\rangle$	$\langle 6, 0\rangle$	$\langle 6, 29\rangle$	8
17	(advance$_2$)	$\langle\{0400\}_o, S_{00}{:}[], \langle S_{02\setminus 0200}, 0210\rangle, \langle S_{01}, 010\rangle\rangle$	$\langle 6, 0\rangle$	$\langle 6, 23\rangle$	8
17	(**Incumbent Update**)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200}, 0210\rangle, \langle S_{01}, 010\rangle\rangle$	$\langle 6, 0\rangle$	$\langle 6, 23\rangle$	20
18	(prune-opt$_1$)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200\setminus 0210}, 0210\rangle, \langle S_{01}, 010\rangle\rangle$	$\langle 6, 0\rangle$	$\langle 6, 23\rangle$	20
18	(advance$_2$)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200\setminus 0210}, 0210\rangle, \langle S_{01}, 0100\rangle\rangle$	$\langle 6, 20\rangle$	$\langle 6, 20\rangle$	20
19	(advance$_1$)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200\setminus 0210}, 0211\rangle, \langle S_{01}, 0100\rangle\rangle$	$\langle 8, 20\rangle$	$\langle 8, 20\rangle$	20
19	(prune-opt$_2$)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200\setminus 0210}, 0211\rangle, \langle S_{01\setminus 0100}, 0100\rangle\rangle$	$\langle 8, 20\rangle$	$\langle 8, 20\rangle$	20
20	(prune-opt$_1$)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200\setminus 0210\setminus 0211}, 0211\rangle, \langle S_{01\setminus 0100}, 0100\rangle\rangle$	$\langle 8, 20\rangle$	$\langle 8, 20\rangle$	20
20	(advance$_2$)	$\langle\{0100\}_o, S_{00}{:}[], \langle S_{02\setminus 0200\setminus 0210\setminus 0211}, 0211\rangle, \langle S_{01\setminus 0100}, 0101\rangle\rangle$	$\langle 8, 16\rangle$	$\langle 8, 16\rangle$	20
21	(terminate$_1$)	$\langle\{0100\}_o, S_{00}{:}[], \bot, \langle S_{01\setminus 0100}, 0101\rangle\rangle$	$\langle\bot, 16\rangle$	$\langle\bot, 16\rangle$	20
21	(prune-opt$_2$)	$\langle\{0100\}_o, S_{00}{:}[], \bot, \langle S_{01\setminus 0100\setminus 0101}, 0101\rangle\rangle$	$\langle\bot, 16\rangle$	$\langle\bot, 16\rangle$	20
22	(schedule$_1$)	$\langle\{0100\}_o, [], \langle S_{00}, 00\rangle, \langle S_{01\setminus 0100\setminus 0101}, 0101\rangle\rangle$	$\langle 0, 16\rangle$	$\langle 22, 16\rangle$	20
22	(terminate$_2$)	$\langle\{0100\}_o, [], \langle S_{00}, 00\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 22, \bot\rangle$	20
23	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00}, 000\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 22, \bot\rangle$	20
24	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00}, 0000\rangle, \bot\rangle$	$\langle 14, \bot\rangle$	$\langle 14, \bot\rangle$	20
25	(prune-opt$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000}, 0000\rangle, \bot\rangle$	$\langle 14, \bot\rangle$	$\langle 14, \bot\rangle$	20
26	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000}, 0001\rangle, \bot\rangle$	$\langle 16, \bot\rangle$	$\langle 16, \bot\rangle$	20
27	(prune-opt$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000\setminus 0001}, 0001\rangle, \bot\rangle$	$\langle 16, \bot\rangle$	$\langle 16, \bot\rangle$	20
28	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000\setminus 0001}, 001\rangle, \bot\rangle$	$\langle 0, \bot\rangle$	$\langle 21, \bot\rangle$	20
29	(advance$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000\setminus 0001}, 0010\rangle, \bot\rangle$	$\langle 15, \bot\rangle$	$\langle 15, \bot\rangle$	20
30	(prune-opt$_1$)	$\langle\{0100\}_o, [], \langle S_{00\setminus 0000\setminus 0001\setminus 0010}, 0010\rangle, \bot\rangle$	$\langle 15, \bot\rangle$	$\langle 15, \bot\rangle$	20
31	(terminate$_1$)	$\langle\{0100\}_o, [], \bot, \bot\rangle$	$\langle\bot, \bot\rangle$	$\langle\bot, \bot\rangle$	20

Figure 7.6: Two worker reduction of the search tree in Figure 7.1, with initial tasks going against the heuristic order. An additional update is made at timestep 17, **after** the one worker reduction. ... represents the reduction from Figure 7.4. **Obj** and **Bound** show the current objective value and bound for the node currently being viewed by each thread. **Best** is the current incumbent objective value.

condition is weaker than that in the literature which requires at least one node that would have been expanded sequentially to be expanded at all times in a parallel run. However, the obvious method of achieving this in practice is to nominate one worker to always strictly follow the sequential search order (excluding eliminated nodes), bringing us back in line with the literature. That is, any time t, $\sigma_{par(1)} \sqsubseteq \sigma_{par(w)}$, where $par(w)$ is a parallel search with w workers.

With a sequentially ordered worker in place, the sequential lower bound property is maintained regardless of the number of parallel workers. This condition does not exclude acceleration anomalies as we only require *at least* the same amount of knowledge at all t.

Non-increasing Runtimes Non-increasing runtimes requires that search time does not increase as we increase the number of workers.

By an inductive argument, based on the sequential lower bound property, for non-increasing runtimes we require, for any time t, $\sigma_{par(w)} \sqsubseteq \sigma_{par(w+1)}$[6]. Assuming all workers operate at the same rate and instant communication between them, we can ensure this with a fixed ordering of parallel tasks. The exact ordering of the parallel tasks is not important so long as it is consistent across runs, i.e. we have a consistent *parallel* heuristic function. Two potential orderings are discussed in Section 7.4.1.

Repeatability Repeatability requires a low variance on search runtimes.

To achieve this we want two identical runs, at any time t to have the same amount of knowledge. In practice differences of knowledge at time t can be caused by parallel overheads such that differences in knowledge form a normal distribution with variance v.

Repeatability is gained as a side effect of the fixed sequential and parallel orderings required for the first two properties as, at any time t, the knowledge should be consistent across runs if we have the number number of workers. Repeatability holds in the anomaly avoidance literature, yet is not commonly discussed.

Taken together the sufficient conditions to achieve replicable search are, at all times t

$$\sigma_{par(1)} \sqsubseteq \sigma_{par(w)} \sqsubseteq \sigma_{par(w+1)}$$

This holds for both decision and optimisation searches.

[6]Again we can weaken this as finding an optimal/target node for w workers before or at the same time as $w + 1$ workers, but this is difficult to guarantee in practice without the stronger ordering.

7.4 Skeletons for Replicable Search

Achieving the replicable search properties requires careful control of both sequential and parallel task ordering to ensure that:

- The sequential order is maintained by at least one worker.
- There is a fixed, but not necessarily sequential, ordering on the parallel tasks.

To allow access to replicable performance, without requiring specialist user knowledge, we have captured these features in a parallel search coordination, the **Ordered** coordination, that gives rise to two additional skeletons: DecisionOrdered and OptimisationOrdered. By having specialised skeletons we allow existing applications, i.e. Node Generators, to get replicable performance guarantees without requiring any changes to the Node Generators themselves.

Ordered adopts a static approach to work generation. As in many static approaches (e.g. Section 2.4.1), Ordered is split into a work generation phase and a parallel search phase.

Pseudocode for the Ordered search coordination is given in Listing 7.1. The `generateWork` function (line 1) is called by a single worker[7] to generate the initial set of tasks. The worker performs a sequential search (i.e. the else clause in line 21) until it reaches the user specified spawn depth (line 7). All node at this depth are added to a list of tasks but are not yet spawned (line 16). The worker backtracks after adding nodes to avoid entering the nodes that will be explored later. After all tasks have been collected, they are assigned fixed parallel priorities (line 36) and only then are they spawned to a global priority workpool (line 38). The actual priorities do not matter so long as they are fixed. We discuss two possible orders in Section 7.4.1.

The `searchPhase` function (line 42) is called by all workers. A single worker is determined to be the sequential worker (line 43) and searches tasks in a left-to-right, i.e. sequential, order. All other workers continuously remove tasks from the global workpool in priority order (line 51) until search completes. To avoid both the sequential and a parallel worker exploring the same sub-tree, we assign a `isStarted` flag to each task (line 46) that is set before searching a particular sub-tree. Updates to the `isStarted` flag are made atomically, as are the `getNextSequential` (line 45) and `getNext` calls (line 51), to ensure workers communicate safely and tasks are not executed twice.

A graphical depiction of Ordered is in Figure 7.7. This shows both the work generation phase that populates the workpool with all nodes at $d_{spawn} = 1$ and the search phase where tasks are removed from both a local, sequential, workpool and the global priority workpool.

[7] As seen in the literature, e.g. EPS [56], work generation can also be done in parallel.

Listing 7.1: Ordered Coordination.

```
function generateWork(SearchType searchType, SearchSpace space, Node root, int spawnDepth):

  // Find all nodes at spawnDepth
  tasks ← []
  GeneratorStack.push(NodeGenerator(space, root))
  while not generatorStack.empty() do
    generator ← generatorStack.top()
    if currentDepth == spawnDepth then
      while generator.hasNext() do
        node ← generator.next()

        // Search type specific node processing is added here
        // This will possibly force a continue instead addending a task to the queue,
        // e.g. on a prune

        // Note: execution of orderedSubtreeSearch is delayed until a later spawn.
        tasks.append(orderedSubtreeSearch(searchType, space, node))
      // Backtrack
      currentDepth ← currentDepth - 1
      generatorStack.pop()
    else
      child ← generator.next()
      if generator.hasNext() then
        node ← generator.next()

        // Search type specific node processing is added here
        // This will possibly force a continue instead pushing children to the stack,
        // e.g. on a prune

        generatorStack.push(NodeGenerator(space, node))
        currentDepth ← currentDepth + 1
      else
        // Backtrack
        currentDepth ← currentDepth - 1
        generatorStack.pop()

    prioritise(tasks) // Assign parallel priorities (see Section 7.4.1)
    for t in tasks:
      spawn t

    return tasks

function searchPhase(SearchType searchType, SearchSpace space, tasks):
  if isSequentialWorker:
    while running:
      t ← getNextSequential(tasks) // Remove task in sequential order
      if not t.isStarted():
        t.isStarted ← true
        sequentialSearch(searchType, space, t.subtreeRoot);
  else:
    while running:
      t ← getNext(tasks) // Remove task in priority order
      if not t.isStarted():
        t.isStarted ← true
        sequentialSearch(searchType, space, t.subtreeRoot);
```

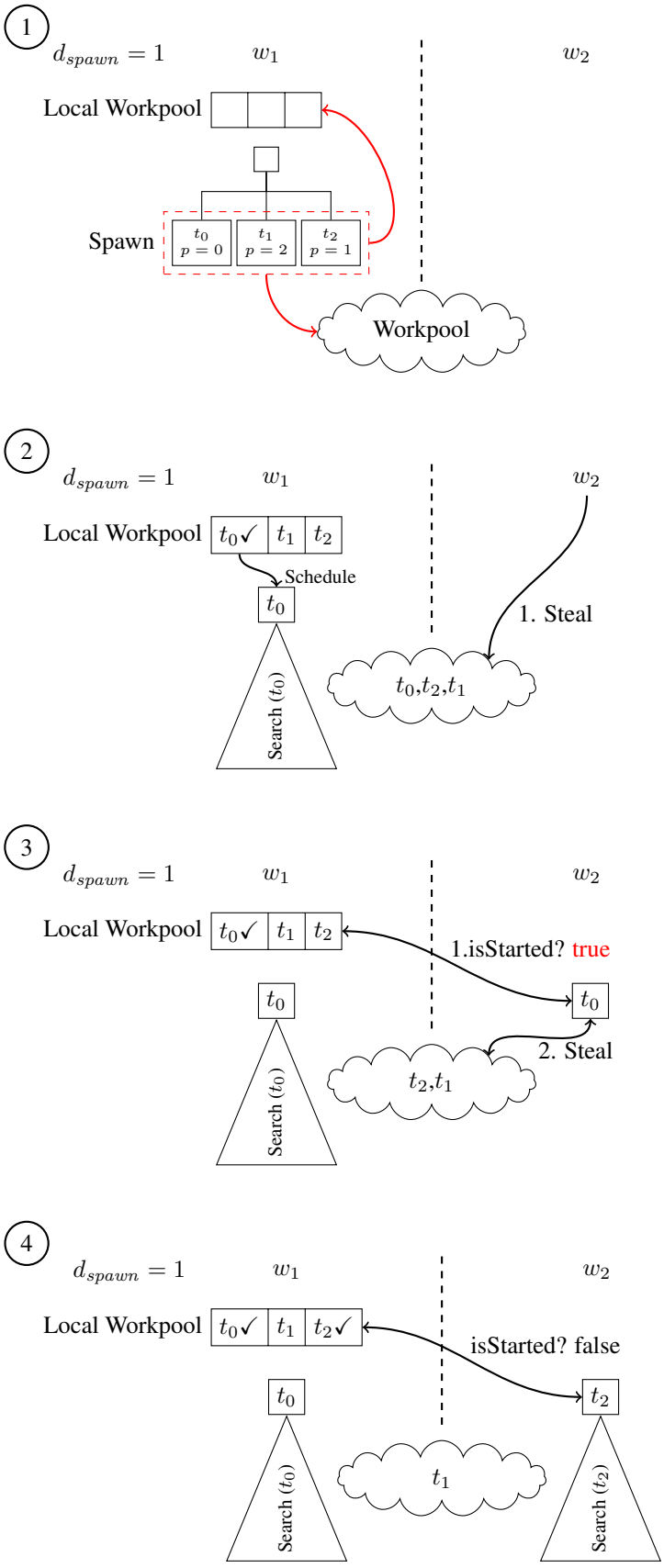


Figure 7.7: Operation of the Ordered search coordination. (1) Worker 1 performs a sequential search to depth d_{spawn} and creates the initial work distribution (both locally and globally). (2) Worker 1 schedules the first (local) task, t_0, while worker 2 attempts to steal from the global workpool. (3) Worker 2 checks if t_0 has already been started on the sequential worker, it has, so worker 2 steals again. (4) Worker 2 checks if t_2 has already been started on the sequential worker, as it has not worker 2 starts searching from t_2.

During work generation the search is divided into many tasks using a user settable spawn depth parameter d_{spawn}. For example, $d_{spawn} = 2$ causes all nodes at depth 2 to be converted into tasks.

In the search phase one worker performs a left-to-right (i.e. sequentially ordered) traversal over the tasks. All other workers perform on-demand work-stealing from the priority workpool. The left-to-right worker ensures the sequential lower bound property is maintained, while the global priority workpool ensures non-increasing runtimes property is met by fixing the parallel ordering. The Ordered coordination is implemented on top of YewPar's Priority Ordered scheduler policy (Section 5.1.3.1) that manages steals from a global priority workpool.

This approach creates **all** tasks during the work generation phase. This is in contrast to Depth-Bounded (Section 4.3.4) that creates new tasks for nodes below d_{cutoff} only when the parent task is *executed*. The requirement to generate work ahead of time is a key limitation as

1. Large[8] values of d_{spawn} can cause the (sequential) work generation phase to dominate runtimes.

2. Large values of d_{spawn} requires large memory requirements to store all tasks in the global workpool. As both the sequential and parallel workers needs access to the tasks some task replication is required, e.g. the two workpools in Figure 7.7. In the worst case the doubles the memory requirements[9].

3. The replicable properties are only valid for a particular d_{spawn} as a fixed parallel workload is required.

4. Pruning decisions happen for trees rooted at d_{spawn}, never a parent node. Consider two sub-trees t_1 and t_2 rooted at d_{spawn} that share a common parent p. In a sequential search p may be pruned causing t_1 and t_2 to never be created. However in Ordered, as work is generated upfront, we must explicitly create, schedule, and prune t_1 and t_2. That is, we may perform more work than a fully sequential search, but not the one worker case (as specified by the properties).

We investigate these limitations empirically in Section 7.5.3.

Additionally, as with Depth-Bounded and Budget, picking an appropriate value for d_{spawn} parameter is difficult and both instance and architecture specific.

[8]The exact value of "large" depends on the instance we are considering.

[9]The current YewPar implementation suffers from this effect, where tasks are essentially stored once in the global priority queue and again in the sequential worker, with only a shared future between them to avoid replicating work. A sufficiently more advanced implementation could enable more sharing to reduce memory requirements.

The Ordered coordination is closely related to existing approaches that use static work generation (Section 2.4.1). Many of these approaches, without necessarily designing for it, achieve replicable performance results. This is due to performing a left-to-right ordering over the set of generated tasks causing at least one worker to always be working on a node in the sequential path. These approaches do not support different parallel orderings as in Ordered.

In some situations the parameter that determines when to stop generating work causes existing approaches to break the replicable performance guarantees. For example, EPS generates work until there is $n \times w$ tasks, where w is the number of workers. This breaks the non-increasing runtime property as the parallel ordering is not fixed as we add more workers.

7.4.0.1 MT^3 Spawn Rule

Unlike the other search coordinations, Ordered does not have a corresponding MT^3 spawn rule as work is generated ahead of time. That is, Ordered is equivalent to a search with initial search state $\langle \{\epsilon\}_o, [S_0, \ldots, S_n], \bot, \bot \rangle$.

Ordered cannot be fully specified in MT^3 as is, due to the choice of always assuming a FIFO based global workpool. In order to allow different sequential and parallel orders the schedule rule would need to allow different workpool access functions. For example:

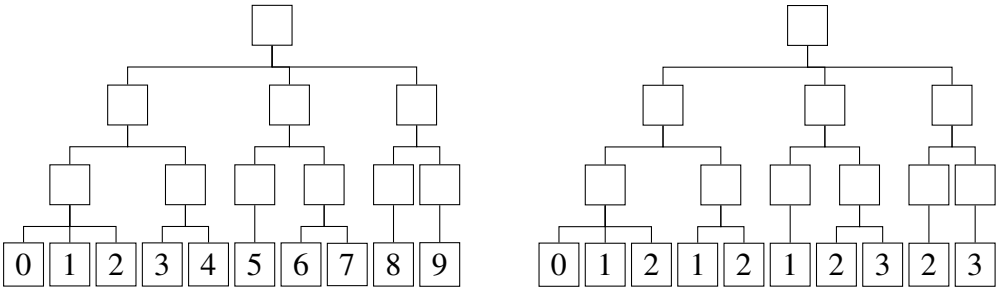
$$(\text{schedule}_i)\ \frac{S = next(Tasks, i) \qquad S \neq \emptyset \qquad v = \text{root of } S}{\langle \sigma, Tasks, \ldots, \bot, \ldots \rangle \rightarrow \langle \sigma, Tasks - S, \ldots, \langle S, v \rangle, \ldots \rangle}$$

By allow `next` function to remove tasks in priority order, we not only allow Ordered to be specified, but also allow for other global search orderings such as best-first search. As we deal solely with depth-first we have chosen not to reflect this change in Chapter 3.

7.4.1 Parallel Task Ordering

To guarantee the replicable properties it is sufficient that the parallel tasks execute in a fixed order, yet the specific order is unimportant. This can be advantageous to introduce diversity into the search by purposefully going against the heuristic order. Figure 7.8 shows two search orders that are currently supported by YewPar.

Figure 7.8(a) assigns priorities in a "left-to-right" fashion. That is, they directly follow the heuristic ordering for the sequential search, essentially looking ahead of the sequential worker. This approach is often used in static work distribution algorithms as it is simple to both implement and reason about.



(a) Linear (heuristic) search order. (b) Discrepancy search order.

Figure 7.8: Example of two possible task orders. Lower is higher priority.

Figure 7.8(b) uses a discrepancy search order. Here priorities are assigned based on the number of discrepancies from the root to that task, i.e. the number of right branches taken [10]. As discussed in Section 2.1.4.1, discrepancy search is based on the idea that search heuristics are often weak near the root of the search tree [18]. Poor heuristic choices are be counteracted by having parallelism go against the heuristics (in increasing number of discrepancies). Other discrepancy orders are possible, for example Archibald et al. [4] use a discrepancy order that also accounts for the depth at which the discrepancy occurs.

7.5 Evaluation of Ordered Skeletons

We evaluate the Ordered skeletons by comparison to the closely related Depth-Bounded skeletons using $d_{spawn} = d_{cutoff}$. As we are trying to show repeatability, results are based on the mean of 30 measurements. Failed runs are ignored.

Three case study applications are used: Subgraph Isomorphism Problem (Section 5.2.2.3), Maximum Clique (Section 5.2.3.1) and the Travelling Salesperson Problem (Section 5.2.3.2). We use a d_{spawn} of 4 for TSP and 2 for Maximum Clique and SIP.

Appendix D shows that there is often little performance difference between the orderings, other than overheads to manage the priority queue for discrepancy search. As the purpose here is to show the using different orderings is possible, so long as they are fixed, rather than to obtain the best performance for each instance, we arbitrarily evaluate Maximum Clique with the discrepancy search order of Figure 7.8 while both SIP and TSP order tasks linearly.

7.5.1 Scaling

Figures 7.9, 7.10 and 7.11 show strong scaling of Maximum Clique, TSP, and SIP, for between 1 and 255 workers (1 to 17 localities). If the sequential lower bound and non-increasing

[10] We ensure that tasks with duplicate priorities are inserted/removed in a fixed order.

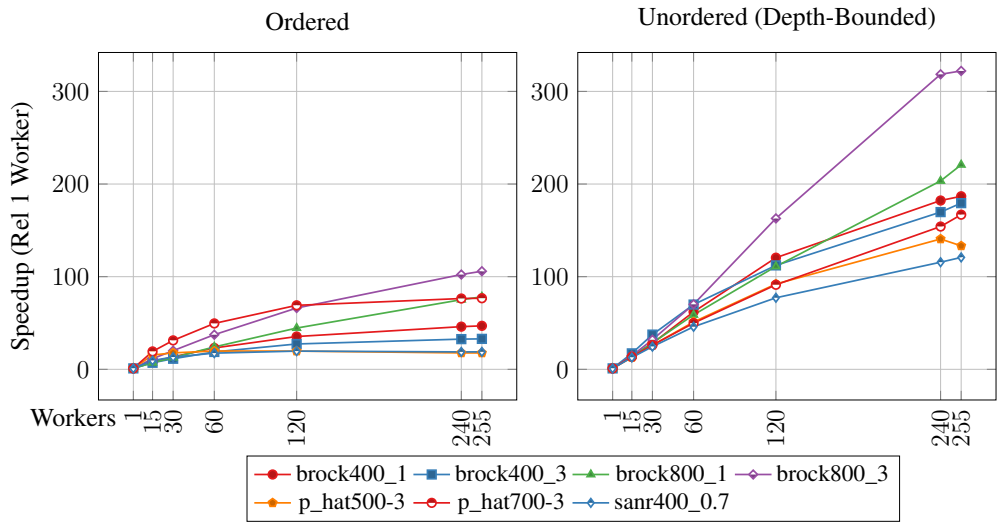


Figure 7.9: Maximum Clique strong scaling. Ordered vs. Depth-Bounded. $d_{spawn} = d_{cutoff} = 2$.

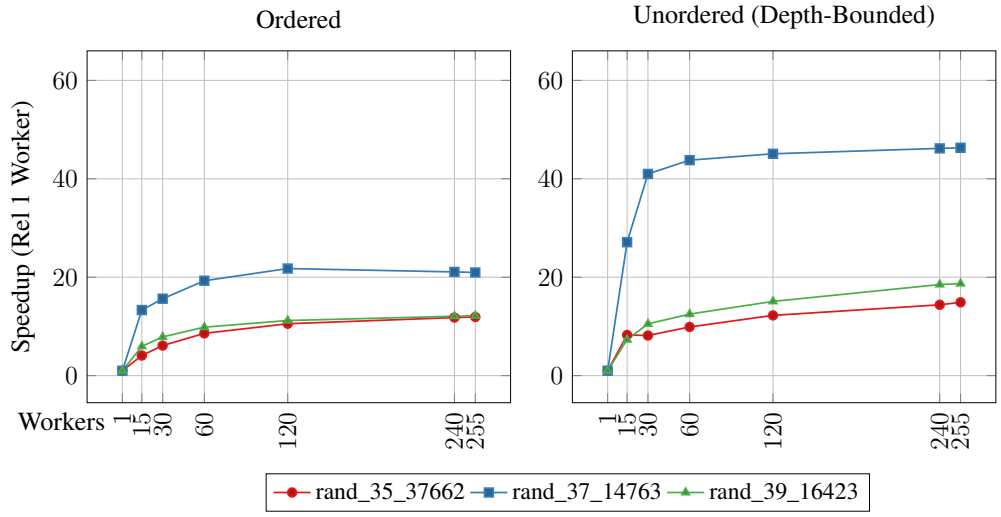


Figure 7.10: TSP strong scaling. Ordered vs. Depth-Bounded. $d_{spawn} = d_{cutoff} = 4$.

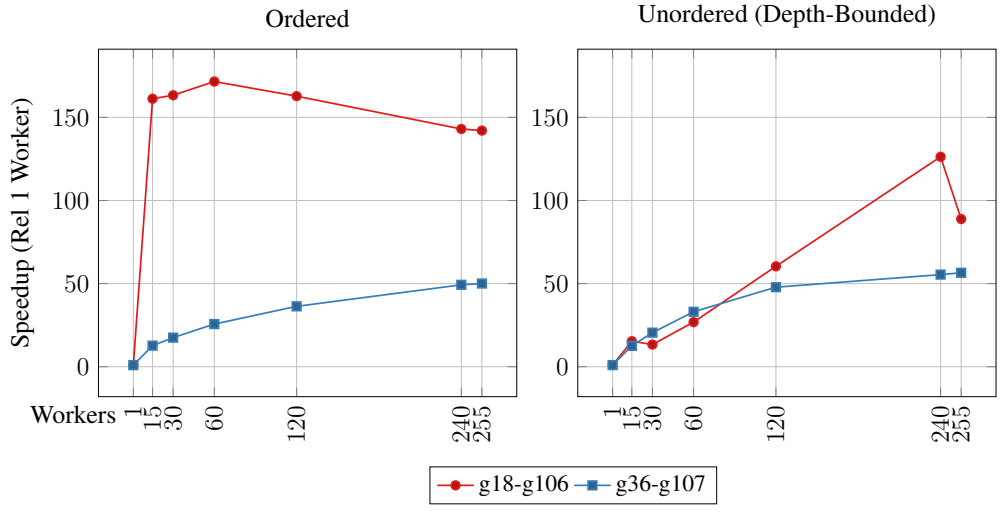


Figure 7.11: SIP strong scaling. Ordered vs. Depth-Bounded. $d_{spawn} = d_{cutoff} = 2$.

Instance	Skeleton	Runtime (s) 1	15	30	60	120	240	255
g18-g106	Depth-Bounded	1,555.32	101.22	116.68	57.91	25.75	12.31	17.50
	Ordered	1,561.31	9.69	9.56	9.10	9.59	10.92	10.99
g36-g107	Depth-Bounded	1,515.06	120.56	73.94	45.82	31.66	27.34	26.81
	Ordered	1,515.09	119.37	86.53	59.03	41.70	30.75	30.27

Table 7.1: (Mean) Runtimes for Subgraph Isomorphism – Ordered and Depth-Bounded. $d_{spawn} = d_{cutoff} = 2$.

runtime properties are maintained then we expect the speedup to always be greater than 1 and non-decreasing. We allow small slowdowns to account for increased parallel overheads (as discussed in Section 7.1).

These properties are maintained for all Maximum Clique and TSP instances. For SIP we observe a slowdown for g18-g106 from 60 to 255 workers. Looking at the runtime data (Table 7.1) we see a runtime difference of less than two seconds between the 60 and 255 worker cases. This is likely caused by the overheads managing the additional 13 localities, rather than a search order effect.

No instances violate the sequential lower bound property even when using Depth-Bounded. It is likely Depth-Bounded maintains a near sequential ordering, even with the effects of random work-stealing, due to the depth-pool (Section 4.4) attempting to maintain a heuristic ordering as much as possible. Another possible reason is that for many applications a good, but not necessarily optimal, bound is often quickly found allowing parallelism to still be effective at pruning areas of the search without perfect knowledge.

For many instances Depth-Bounded also appears to achieve non-decreasing runtime property. This is however due to an averaging effect. If we consider all samples for a specific instance, e.g. brock800_3 in Figure 7.12, we see that the non-decreasing runtime property is broken. That is, we can draw a line between any two scaling points giving many opportunities for the Depth-Bounded to break the property. On the other hand the limited variability of Ordered shows that non-decreasing runtimes in all cases, even over multiple runs.

Depth-Bounded scales better than Ordered in general, with maximum scaling in Maximum Clique for the Ordered case ranging between 0-100 compared to 100-200 for Depth-Bounded. Looking at the runtime figures of Table 7.2 we see that absolute differences are often relatively small, e.g. around 6 seconds for the brock400 instances at 255 workers. For larger instances such as the brock800 series there is wider absolute difference in running time likely caused, in part, by the increased time spent in (sequential) work generation. Percentage slowdown is often high, with Ordered being around 40% slower than Depth-Bounded over all instances and scales, and 72% slower for 255 workers over all instances. These overheads are quantified further in Section 7.5.3.

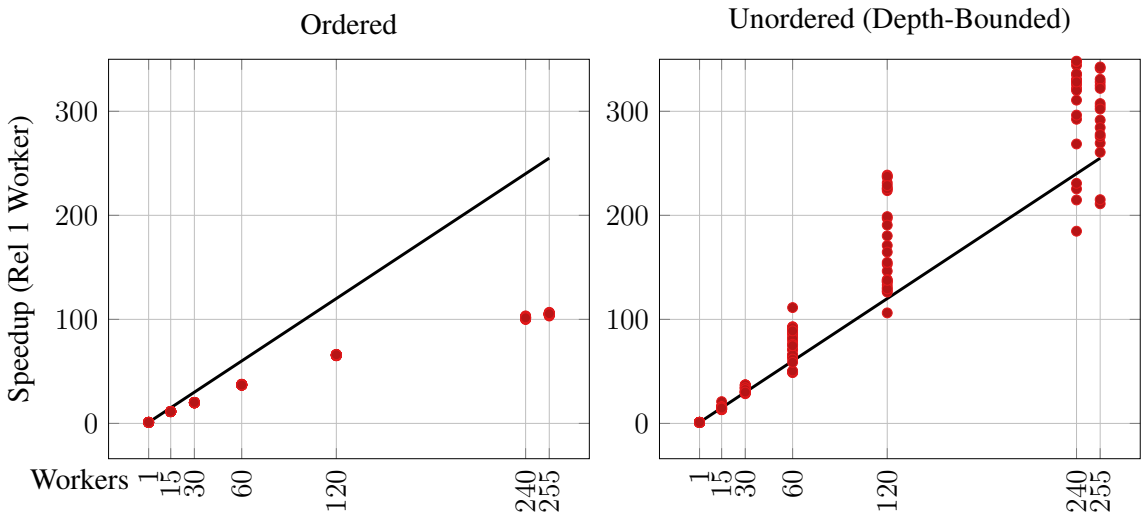


Figure 7.12: Distribution of scaling results for brock800_3.

Travelling Salesperson scales poorly for both Ordered and Depth-Bounded. This is likely due to the low value of d_{spawn} and d_{cutoff}. As seen in Section 6.5 scaling improves for TSP as d_{cutoff} increases. Unfortunately, due to the limitations of Ordered, quantified in Section 7.5.3 it is not possible to increase the spawn depth without incurring large memory and runtime overheads during the work generation phase.

One worker runtimes for Ordered are often better than that of Depth-Bounded. This is expected as the sequential worker requires no parallel scheduling loop whereas Depth-Bounded always spawns and removes tasks from the workpool even in the one worker case.

7.5.2 Repeatability

To show the third property, repeatability, we use relative standard deviation (RSD) as a measure of sample dispersion. Figure 7.13 shows RSD plotted as a cumulative probability distribution for Maximum Clique, TSP and SIP. The distribution is created by combining the RSD from *all* instances at each number of workers[11]. The further to the left that the probability function reaches 1, the more repeatable.

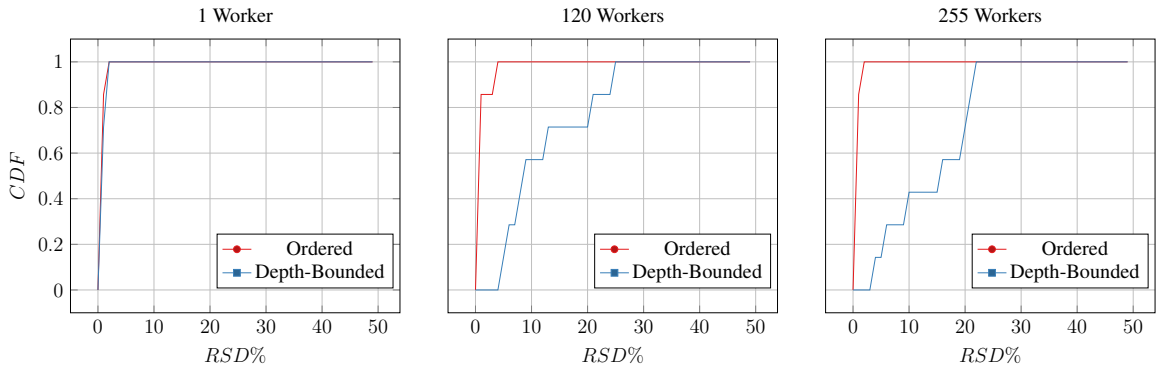
Overall Ordered is more repeatable than Depth-Bounded.

For single worker cases repeatability is high for both skeletons due to the lack of ordering effects. More surprisingly, for workers > 1, as we increase the number of workers the repeatability does not significantly degrade even for Depth-Bounded. This is potentially caused by the low average runtimes for high worker counts giving less room for variation.

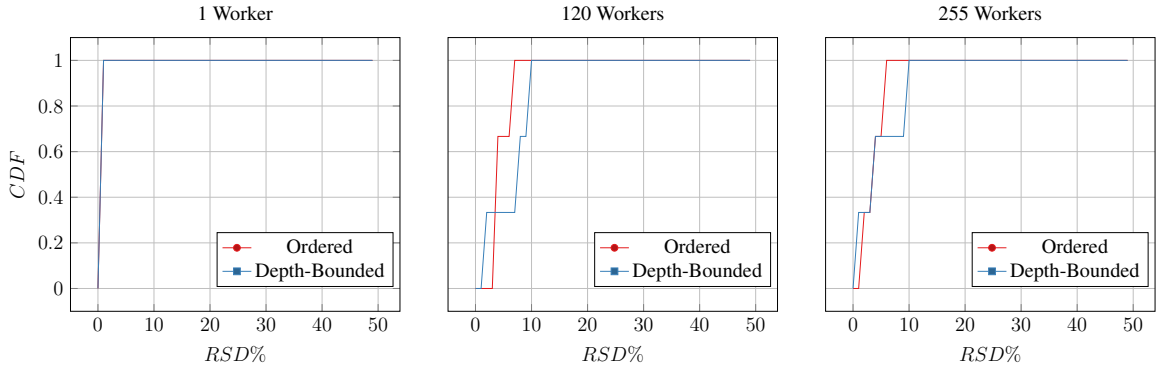
[11]Unfortunately, as seen in Archibald et al.[4], RSD is not robust to outliers. However the effect of outliers appears limited in this work.

Instance	Skeleton	Runtime (s) 1	15	30	60	120	240	255	Mean
	Depth-Bounded	419.35	30.58	14.95	6.76	3.48	2.30	2.25	
brock400_1	Ordered	417.41	51.84	31.02	18.11	11.79	9.06	8.89	
	Slowdown (%)	−0.46	41.01	51.82	62.66	70.48	74.61	74.76	51.16
	Depth-Bounded	244.78	14.22	6.53	3.50	2.18	1.44	1.36	
brock400_3	Ordered	243.51	34.40	21.35	12.98	8.90	7.47	7.41	
	Slowdown (%)	−0.52	58.65	69.40	73.01	75.49	80.70	81.58	59.82
	Depth-Bounded	5,270.80	366.72	185.25	89.68	47.47	25.92	23.88	
brock800_1	Ordered	5,238.76	733.64	413.92	216.99	117.63	69.71	66.97	
	Slowdown (%)	−0.61	50.01	55.24	58.67	59.65	62.82	64.35	48.18
	Depth-Bounded	4,891.09	334.74	153.24	69.71	30.03	15.36	15.19	
brock800_3	Ordered	4,873.84	425.10	242.16	130.44	73.67	47.64	46.03	
	Slowdown (%)	−0.35	21.26	36.72	46.56	59.24	67.76	67.01	40.49
	Depth-Bounded	154.05	11.95	5.93	3.01	1.67	1.10	1.16	
p_hat500-3	Ordered	153.64	9.97	8.66	7.78	7.82	8.67	8.70	
	Slowdown (%)	−0.27	−19.86	31.48	61.30	78.60	87.38	86.70	40.09
	Depth-Bounded	1,544.75	113.78	59.20	30.97	16.93	10.02	9.26	
p_hat700-3	Ordered	1,539.00	79.73	49.29	31.06	22.28	20.14	20.06	
	Slowdown (%)	−0.37	−42.71	−20.11	0.30	24.03	50.24	53.86	3.95
	Depth-Bounded	114.43	8.97	4.68	2.49	1.48	0.99	0.95	
sanr400_0.7	Ordered	113.43	11.90	8.57	6.49	5.77	6.03	6.02	
	Slowdown (%)	−0.88	24.65	45.40	61.62	74.31	83.60	84.26	50.04
Mean	Slowdown (%)							72.85	40.83

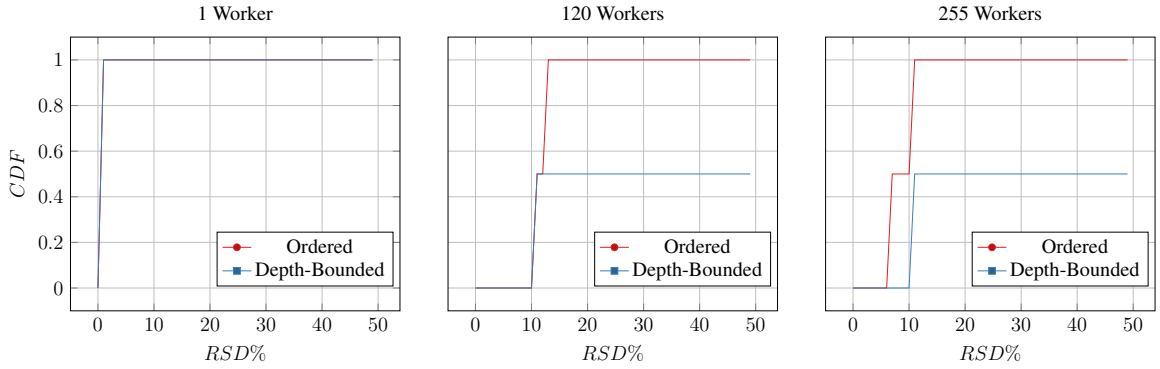
Table 7.2: (Mean) Runtimes for Maximum Clique – Ordered and Depth-Bounded. $d_{cutoff} = d_{cutoff} = 2$. (geometric) Mean slowdowns are reported over all scales and over all instances.



(a) Maximum Clique repeatability: Ordered vs. Depth-Bouned.



(b) TSP repeatability: Ordered vs. Depth-Bounded.



(c) SIP repeatability: Ordered vs. Depth-Bounded.

Figure 7.13: Repeatability of Ordered and Depth-Bounded.

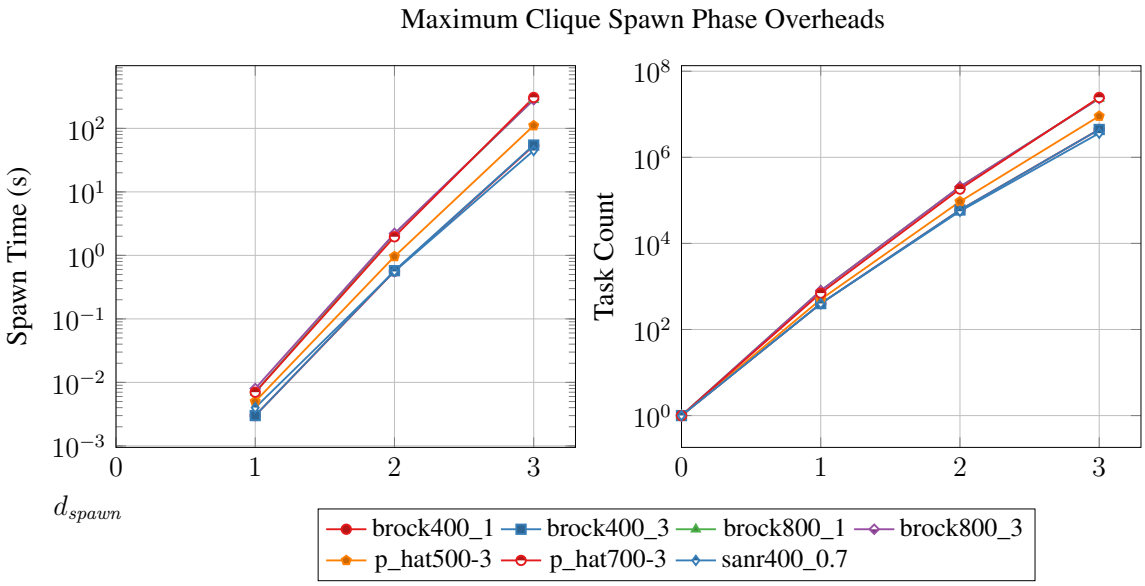


Figure 7.14: Maximum Clique: impact of increasing d_{spawn} on Ordered. Note the logarithmic scale.

For Maximum Clique (Figure 7.13(a)), Ordered achieves significantly better RSD values, less than 4% in all cases compared to greater than 20% for Depth-Bounded. Travelling Salesperson (Figure 7.13(b)), likely due to poor scaling of TSP at low d_{spawn} / d_{cutoff}, achieves similar levels of repeatability for both Ordered and Depth-Bounded.

For Subgraph Isomorphism (Figure 7.13(c)) while the RSD of Ordered increases to just over 10%, Depth-Bounded is significantly less repeatable showing an RSD of greater than 50% in all cases and a maximum 191% RSD. The increased RSD for Depth-Bounded is caused by SIP being more sensitive to search order as it determines early termination. Ordered, by carefully controlling the search order, ensures that, if a solution is found early in the search, it is always found early.

7.5.3 Limitations

The key limitation of Ordered is that all tasks are spawned ahead of time the (sequential) work generation phase. This incurs both time, to generate the tasks, and memory, to store the tasks on a single locality, overheads. Figure 7.14 shows both the time to spawn and number of tasks spawned for the Maximum Clique instances[12]. As we might expect, due to the combinatorial nature of Maximum Clique, both the spawn time and number of tasks rise exponentially (note the logarithmic y-axis).

No instance managed to spawn work with d_{spawn} greater than 3 in less than 10 minutes, even for small instances (e.g. brock400_3). Putting this in context, the runtime of brock400_3

[12]Based on a single run. Task counts remain constant on repeated runs but runtimes may vary.

using Sequential is just over 7 minutes (Section 6.3). This means the Ordered skeleton takes longer to generate work for a parallel search than it would to simply run search sequentially. For $d_{spawn} = 3$ the spawn time for brock400_3 is 54s, around 14% of sequential search time.

The memory requirements likewise increase exponentially. Assuming a node for Maximum Clique takes around 160 bytes of memory[13] then for brock400_1 at $d_{spawn} = 3$ we already require around 0.5GB to store the initial tasks. In the implementation there is no sharing of nodes between the sequential worker and parallel workers, requiring at least 1GB of memory. This effect is significantly worse for larger instances such as brock800_1 that requires more the 5GB of memory at $d_{spawn} = 3$.

Finally, while using a low value d_{spawn} does not seem to be a problem for Maximum Clique, Section 6.5 suggests that for applications such as TSP splitting the work low in the tree can cause poor parallel performance. To scale Ordered to higher worker counts, future work is required to parallelise the work generation phase, support better sharing between the sequentially ordered worker and other workers, and support a distributed-memory, priority-ordered, workpool.

7.6 Summary

Being able to reason about the parallel performance of branch and bound searches is essential for domains such as empirical algorithmic design. A search is performance replicable if it guarantees the following three properties:

Sequential Lower Bound: Parallel runtime is never higher than sequential (one worker) runtime.

Non-increasing Runtimes: Parallel runtime does not increase as the number of workers increases.

Repeatability: Parallel runtimes of repeated searches on the same parallel configuration have low variance.

Search anomalies are caused by search ordering effects and are central to understanding the conditions where these properties are broken. We have seen how MT^3 (Chapter 3) can be used to extend the existing literature on anomaly-avoiding search by extending MT^3 to support an ordering on search states σ (Section 7.3). The properties may be achieved by ensuring for all time t:

$$\sigma_{par(1)} \sqsubseteq \sigma_{par(w)} \sqsubseteq \sigma_{par(w+1)}$$

[13] Two, 4 byte integers, a 112 byte bitset and a vector we assume requires around 40 bytes at low d_{spawn}.

Where $\sigma_{par(w)}$ is the search state at time t with w workers. This result holds for both decision and optimisation searches.

To make it easy for non-experts to benefit from replicable search, we have designed and implemented specialised search skeletons, the Ordered skeletons (Section 7.4). Ordered guarantees that at least one worker is searching in the sequential order while allowing all other workers to explore the search in any *fixed* parallel order (Section 7.4.1). This is achieved by using upfront work generation and a global priority-based workpool. Ordered allows existing user applications, i.e. Node Generators, to gain performance guarantees without application changes other than enabling the skeleton.

Ordered has been evaluated on a mix of both optimisation and decision problems (Section 7.5) showing that, while the scaling performance is often worse than Depth-Bounded (Section 4.3.4), the sequential lower bound and non increasing runtime properties are maintained in all cases. Ordered is significantly more repeatable than Depth-Bounded for Maximum Clique and SIP while, due to a lack of scaling, maintaining similar RSD values for TSP.

A detailed look at the limits of the Ordered skeleton shows (Section 7.5.3) significant overheads, of time and memory, as the (static) spawn depth, d_{cutoff} increase. Future work is required to overcome these limitations.

Chapter 8

Conclusion

Exploiting the ubiquity of parallel hardware is important for reducing existing exact combinatorial search search runtimes and allowing larger instances to be solved. However, there are many challenges to overcome including managing the highly irregular search tree shape and ensuring search heuristics are maintained as much as possible. These challenges mean many search problems are not parallelised at all, or parallelised on an ad-hoc basis for a particular application and scale, and not designed to be reusable.

This thesis investigates the potential of a unified approach to exact parallel combinatorial search that works at every scale (from desktop to large cluster, cloud, or HPC), while removing the burden of parallel programming from the search domain expert. It does so by presenting parallel algorithmic skeletons for exact combinatorial search problems as a reusable, domain-independent, parallelism approach.

8.1 Summary

Chapter 2 provides background information on exact combinatorial searches which solve problems using backtracking search algorithms. By parallelising the backtracking search we achieve search independent, and hence widely applicable, search parallelisations.

The overview of parallel search in Section 2.2 describes three main methods for introduce parallelism to search: parallel node processing, where branching/bounding operations are performed in parallel; space-splitting, where sub-trees of the main search tree are explored in parallel; and portfolio approaches that run multiple full searches in parallel. We focus on space-splitting as it is both commonly used in existing parallelism approaches and is domain-independent.

Existing space-splitting parallelism approaches are critically reviewed in Section 2.4 where we show that they may be categorised into **static** approaches, where a fixed workload is

determined ahead of search, and **dynamic** approaches, that can generate new tasks at runtime as required. Static approaches influence the design of the Depth-Bounded (Section 4.3.4) and Ordered (Section 7.4) skeletons, while dynamic approaches based on random work-stealing influences Stack-Stealing (Section 4.3.5), and periodic load-balancing influences Budget (Section 4.3.6).

Many existing parallel search approaches are not designed with reuse in mind. Furthermore, although there are many existing task-parallel frameworks they are not appropriate for search, e.g. they steal work from workpools instead of generating work dynamically and they often break heuristic search orders. Given this we motivate the need for a new framework in Section 2.4.4.

Chapter 3 presents a novel formal model, MT^3, for general parallel backtracking search problems. The model, based on operational semantics, allows for precise specification of parallel backtracking search. The model informs the design of an abstract framework that the skeletons are designed against (Section 4.2), succinctly describes the operation of the Depth-Bounded, Stack-Stealing, and Budget search coordinations (Sections 4.3.4.1, 4.3.5.1 and 4.3.6.1), and shows how performance anomalies affect parallel search (Section 7.2).

In addition the model forms the basis of a suitably generic programming interface for the skeletons: Lazy Node Generators (Section 3.4). Lazy Node Generators are a uniform abstraction for application developers to specify how application-specific search trees are created, including implicitly encoding application-specific search order heuristics. Search tree nodes are constructed lazily, allowing pruning to eliminate redundant computation. That is, it eliminates sub-trees before they manifest based on shared search knowledge. We show the generality of the Lazy Node Generator programming interface by specifying the search trees of seven applications (Section 5.2).

Chapter 4 describes a set of general-purpose algorithmic skeletons for search. The skeletons are parameterised by Lazy Node Generators allowing a user to provide domain-specific functionality. The skeletons are general enough to allow the three different types of search (enumeration, decision, and optimisation) to utilise the same parallel search coordinations. For scalability, the skeletons target distributed-memory architectures. Four search coordinations are described: Sequential (Section 4.3.3), which generates a single search task and performs a depth-first search; Depth-Bounded (Section 4.3.4), which causes any node above a user specific d_{cutoff} to be converted to a parallel task; Stack-Stealing (Section 4.3.5), which allows workers to directly request work from each other; and Budget (Section 4.3.6), which spawns work after a user specified number of backtracks has been performed. Reusability of the skeletons is demonstrated by applying the skeletons to seven different search applications (Section 5.2).

Chapter 5 introduces YewPar, a new parallel search framework that implements the skeletons of Chapter 4. YewPar is written in C++ and utilises HPX [28] to provide distributed-memory support for task-parallelism. It provides both the Lazy Node Generator interface (Section 3.4) and parallel skeleton implementations (Chapter 4), as well as components such as custom work-stealing scheduling and global knowledge management, e.g. for incumbents. The custom work-stealing scheduling is required due to limitations with existing work-stealing approaches, namely that they often break search order heuristics (Section 4.4).

Seven applications, covering a range of enumeration, decision, and optimisation problems, are used to evaluate the skeletons: counting the number of numerical semigroups of a particular genus (Section 5.2.1.2), the synthetic Unbalanced Tree Search benchmark (Section 5.2.1.1), k-Clique applied to a finite geometry case study (Section 5.2.2.1), the Subgraph Isomorphism Problem (Section 5.2.2.3), finding Maximum Cliques in graphs (Section 5.2.3.1), finding shortest path Travelling Salesperson tours (Section 5.2.3.2), and finding an optimal packing in 0/1 Knapsack (Section 5.2.3.3). The applications themselves are tested using more than 25 problem instances showing the wide applicability of the approach.

Chapter 6 uses YewPar to systematically analyse the performance of the skeletons leading to the following key conclusions:

1. Across 21 instances of Maximum Clique we see a mean slowdown of 6.1%, with maximal slowdown of 12.6%, when comparing the skeleton approach to hand-written searches (Section 6.3). These slowdowns are attributable to the cost of additional node copies that can be avoided in hand-written searches by updating nodes in place.

2. By recording the incumbent update times of Maximum Clique instances, we show the use of a partitioned global address space (PGAS) and bounds broadcasting for branch-and-bound knowledge exchange is appropriate as, in general, there is a relatively small number of total update messages (Section 6.4). Techniques such as pre-initialising bounds can be used to further reduce the number of updates required if necessary.

3. Depth-Bounded, despite the simplicity of the approach, can achieve good parallel performance for most case studies (Section 6.5), including an average best case maximum speedup of 89 for Maximum Clique and 24 across all applications/instances on 120 workers (Section 6.5). Choosing the value of d_{cutoff} remains a challenge, with the best value varying widely over all applications. Choosing it incorrectly can lead to average speedups over all applications of less than 2 on 120 workers (Section 6.8).

4. Comparing deque-based and depth-pool based work-stealing shows that both achieve similar performance. Given this, we advocate the use of the depth-pool as a more principled, yet similarly performing, workpool structure.

5. Stack-Stealing has the advantage of not requiring tuning and achieves good parallel performance for most case studies (Section 6.6). The chunking optimisation for Stack-Stealing, where multiple nodes are returned on a steal request, does not help performance (Section 6.6.1). On average, Stack-Stealing performs best out of all the skeletons showing 37 times speedup on 120 workers, including an average best case maximum speedup of 91 for SIP (Section 6.8). However, Stack-Stealing does not necessarily perform best for specific applications.

6. Budget likewise achieves good parallel performance (Section 6.7), including an average best case maximum speedup of 86 for UTS and 34 across all applications/instances on 120 workers (Section 6.8). A surprising result is that a budget of 10^5 backtracks appears to work well, yet not necessarily the best, across all of the case studies.

7. The skeletons scale well[1] for a selection of larger instances on 255 workers, often achieving a (geometric) mean efficiency of greater than 60% and maximum efficiency of 112% relative to the 15 worker case (Section 6.9). Runtimes continue to improve even for 255 workers, suggesting the skeletons will scale further if resources are available.

Chapter 7 designs a specialised skeleton, Ordered, for providing replicable performance in branch and bound searches. Replicable performance is defined as a search with the following performance properties:

Sequential Lower Bound: Parallel runtime is never higher than sequential (one worker) runtime.

Non-increasing Runtimes: Parallel runtime does not increase as the number of workers increases.

Repeatability: Parallel runtimes of repeated searches on the same parallel configuration have low variance.

These properties can be broken due to search anomalies (Section 2.3.2.1). Using MT^3 we have shown how search anomalies can occur (Section 7.2) and that by carefully fixing the ordering of tasks the properties can be maintained. Empirical analysis shows that the properties are maintained in all cases (Section 7.5), however the performance of Ordered is shown to be is 41% slower on average than Depth-Bounded (73% worst case) for Maximum Clique. For SIP, Ordered successfully maintains a relative standard deviation (RSD) of less than 15% in all cases while Depth-Bounded suffers from an RSD of greater than 50%, showing the importance of carefully controlling search orders for repeatability (Section 7.5).

[1]Depth-Bounded fails to parallelise Numerical Semigroups and is excluded from these average results.

In summary, this thesis has demonstrated the feasibility of distributed-memory algorithmic skeletons as a practical and general purpose means of undertaking exact combinatorial search at scale.

8.2 Future Directions

There are many avenues to further develop this work, including:

Integration with existing frameworks: The skeletons are designed to be used in a domain-independent manner. This makes it possible to use YewPar as a parallel coordination layer for existing search frameworks, such as Gecode [162], rather than only for custom search applications. While many static parallelism approaches (Section 2.4.1) allow existing solvers to be used, dynamic solvers often do not as it requires tracking the search internally in order to be able to perform dynamic work-splitting. Further work is required to investigate this integration further, including ensuring the Lazy Node Generator API is sufficiently general and to provide additional skeleton APIs to aid integration.

Supporting rich search techniques: The skeletons are based around a core set of classic and relatively simple search algorithms, i.e. backtracking search, possibly with branch and bound. More recent search algorithms are rich in that they support features such as restarts [163] and no-good recording [164]. Many of these techniques have been explored in the SAT and Constraint Programming communities and future work is required to bring these into a general-purpose framework. This includes determining a suitable domain-independent programming interface than allows a general-purpose framework to use these techniques. Recent unpublished work suggests that techniques such as restarts are not only beneficial for search, but can also be used to remove irregularity from the parallelism by bounding maximum task running time. Such techniques are most commonly used for decision and optimisation searches and it is unclear how they map to enumeration problems as the same part a search space may be traversed more than once. This needs to be carefully controlled to avoid double enumerations.

Improved work-stealing for scale: The Unbalanced Tree Search benchmark (UTS) is commonly used as an example irregular workload for testing new load-balancing algorithms, e.g. [125, 126]. It remains an open question how well UTS represents non-synthetic search trees, or the set of parameters required to make it do so. If UTS is a good representation of real world search trees then a future direction is to apply these new load-balancing techniques to different application domains. The skeletons presented

in this work, by hiding the implementation details, makes it possible to trial new load-balancing transparently, without changes to a users application code.

Automatically deducing parallelism parameters: A major downside of the skeletons in their current form is the requirement for a user to manually specify appropriate tuning parameters, e.g. d_{cutoff} and $budget$. Given multiple skeletons, which should the user choose for maximum performance? Automated approaches to choosing appropriate parameters is a promising research direction and one that is beginning to be explored by the parallel search community, for example using the SMAC tool [165]. Parameter tuning is further complicated when trying to run parallel searches at scale as we may not have access to the final hardware environment, e.g. due to supercomputing budgets, in order to perform ahead of time turning.

Investigating multi-parallelism approaches: Currently, by selecting a specific skeleton, the user chooses a single style of parallelism for the application. An interesting research direction is whether it is beneficial to allow multiple types of parallelism within a search, for example, applying a static approach at the beginning of search, moving to a budget based approach within each static task to identify large tasks, and then random work-stealing near the end of search to quickly process these tasks. Such an approach can be implemented transparently in the skeleton framework. This is different in style to portfolio approaches in that this runs one search with three types of parallelism, wheras portfolios would run three searches with one type of parallelism.

Increased Scale: A key goal of the skeletons is to provide a unified approach to parallelism that works from multi-core to HPC scales. Currently the skeletons have only been tested on a medium sized cluster (255 workers), although the HPX framework underpinning YewPar has been shown to be effective on large HPC setups [120]. Further work is required to ensure transparent scalability. For example algorithms could adapt to be better suited to multi-core, where steals are cheap, or cloud environments, where due to the shared nature of the resources networking performance cannot be guaranteed. Heterogenous architectures are becoming increasingly common and require careful management, particularly for static approaches.

Bibliography

[1] Vangelis T. Paschos. *Applications of combinatorial optimization*. 2nd. London: ISTE, 2014 (cit. on p. 1).

[2] Oded Goldreich. *P, NP, and NP-completeness: the basics of computational complexity*. Cambridge, NY: Cambridge University Press, 2010 (cit. on pp. 1, 9).

[3] Christian Blum and Andrea Roli. "Metaheuristics in combinatorial optimization: Overview and conceptual comparison". In: *ACM Comput. Surv.* 35.3 (2003), pp. 268–308. DOI: 10.1145/937503.937505 (cit. on pp. 1, 11).

[4] Blair Archibald et al. "Replicable parallel branch and bound search". In: *J. Parallel Distrib. Comput.* 113 (2018), pp. 92–114. DOI: 10.1016/j.jpdc.2017.10.010 (cit. on pp. 5, 41–43, 45, 62, 94, 95, 158, 173, 176).

[5] Blair Archibald et al. "Towards Generic Scalable Parallel Combinatorial Search". In: *Proceedings of PASCO 2017*. PASCO '17. Kaiserslautern, Germany: ACM, 2017. DOI: 10.1145/3115936.3115942 (cit. on pp. 5, 65, 67, 94, 110, 151).

[6] Ciaran McCreesh. "Solving hard subgraph problems in parallel". PhD thesis. University of Glasgow, 2017 (cit. on p. 5).

[7] Florent Hivert. *Numeric Monoid Source Repository*. https://github.com/hivert/NumericMonoid. 2018 (cit. on pp. 5, 108).

[8] Matthew C. Schmidt et al. "A scalable, parallel algorithm for maximal clique enumeration". In: *J. Parallel Distrib. Comput.* 69.4 (2009), pp. 417–428. DOI: 10.1016/j.jpdc.2009.01.003 (cit. on pp. 8, 12, 33, 37).

[9] Coenraad Bron and Joep Kerbosch. "Finding All Cliques of an Undirected Graph (Algorithm 457)". In: *Commun. ACM* 16.9 (1973), pp. 575–576 (cit. on p. 9).

[10] Patrick Prosser. "Exact Algorithms for Maximum Clique: A Computational Study". In: *Algorithms* 5.4 (2012), pp. 545–587. DOI: 10.3390/a5040545 (cit. on pp. 9, 109).

[11] Gerhard J. Woeginger. "Exact Algorithms for NP-Hard Problems: A Survey". In: *Combinatorial Optimization - Eureka, You Shrink!, Papers Dedicated to Jack Edmonds, 5th International Workshop, Aussois, France, March 5-9, 2001, Revised Papers*. 2001, pp. 185–208 (cit. on p. 9).

[12] João P. Marques Silva, Inês Lynce, and Sharad Malik. "Conflict-Driven Clause Learning SAT Solvers". In: *Handbook of Satisfiability*. IOS press, 2009, pp. 131–153. DOI: 10.3233/978-1-58603-929-5-131 (cit. on p. 11).

[13] Jordan Bell and Brett Stevens. "A survey of known results and research areas for n-queens". In: *Discrete Mathematics* 309.1 (2009), pp. 1–31. DOI: 10.1016/j.disc.2007.12.043 (cit. on p. 12).

[14] Armin Biere et al., eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009 (cit. on p. 12).

[15] Toshihide Ibaraki. "Theoretical comparisons of search strategies in branch-and-bound algorithms". In: *International Journal of Parallel Programming* 5.4 (1976), pp. 315–344. DOI: 10.1007/BF00998631 (cit. on p. 13).

[16] Francesca Rossi, Peter van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming*. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006 (cit. on p. 14).

[17] Robert M. Haralick and Gordon L. Elliott. "Increasing Tree Search Efficiency for Constraint Satisfaction Problems". In: *Artif. Intell.* 14.3 (1980), pp. 263–313. DOI: 10.1016/0004-3702(80)90051-X (cit. on p. 14).

[18] William D. Harvey and Matthew L. Ginsberg. "Limited Discrepancy Search". In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*. 1995, pp. 607–615 (cit. on pp. 14, 130, 173).

[19] Toby Walsh. "Depth-bounded Discrepancy Search". In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*. 1997, pp. 1388–1395 (cit. on p. 14).

[20] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* (30.3 2005), pp. 202–210 (cit. on p. 15).

[21] The MPI Forum. *MPI-2: Extensions to the Message-Passing Interface*. Tech. rep. University of Tennessee, Knoxville, July 1997 (cit. on p. 16).

[22] B. Ramakrishna Rau and Joseph A. Fisher. "Instruction-level parallel processing: History, overview, and perspective". In: *The Journal of Supercomputing* 7.1-2 (1993), pp. 9–50. DOI: 10.1007/BF01205181 (cit. on p. 16).

[23] Chris Lomont. *Introduction to Intel Advanced Vector Extensions. Intel White Paper*. 2011 (cit. on p. 17).

[24] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492 (cit. on pp. 17, 27).

[25] Matei Zaharia et al. “Apache Spark: a unified engine for big data processing”. In: *Commun. ACM* 59.11 (2016), pp. 56–65. DOI: 10.1145/2934664 (cit. on pp. 17, 27).

[26] Charles E. Leiserson. “The Cilk++ concurrency platform”. In: *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*. 2009, pp. 522–527. DOI: 10.1145/1629911.1630048 (cit. on pp. 17, 108).

[27] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O’Reilly Media, Inc.", 2007 (cit. on p. 17).

[28] Hartmut Kaiser et al. “HPX: A Task Based Programming Model in a Global Address Space”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, OR, USA, October 6-10, 2014*. 2014, 6:1–6:11. DOI: 10.1145/2676870.2676883 (cit. on pp. 17, 95, 185).

[29] Barbara Liskov and Liuba Shrira. “Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems”. In: *Proceedings of the ACM SIGPLAN’88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. 1988, pp. 260–267. DOI: 10.1145/53990.54016 (cit. on pp. 17, 70).

[30] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. “Load distributing for locally distributed systems”. In: *Computer* 25.12 (1992), pp. 33–44 (cit. on p. 18).

[31] Yu-Kwong Kwok and Ishfaq Ahmad. “Static scheduling algorithms for allocating directed task graphs to multiprocessors”. In: *ACM Comput. Surv.* 31.4 (1999), pp. 406–471. DOI: 10.1145/344588.344618 (cit. on p. 18).

[32] L.V. Kalé and S. Krishnan. “CHARM++: A Portable Concurrent Object Oriented System Based on C++”. In: *Proceedings of OOPSLA’93*. Ed. by A. Paepcke. ACM Press, September 1993, pp. 91–108 (cit. on p. 18).

[33] Jixiang Yang and Qingbi He. “Scheduling Parallel Computations by Work Stealing: A Survey”. In: *International Journal of Parallel Programming* 46.2 (2018), pp. 173–197. DOI: 10.1007/s10766-016-0484-8 (cit. on p. 19).

[34] Robert D. Blumofe et al. “Cilk: An Efficient Multithreaded Runtime System”. In: *J. Parallel Distrib. Comput.* 37.1 (1996), pp. 55–69. DOI: 10.1006/jpdc.1996.0107 (cit. on pp. 19, 33, 87).

[35] James Dinan et al. “Scalable work stealing”. In: *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. 2009. DOI: 10.1145/1654059.1654113 (cit. on pp. 19, 86).

[36] R. Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000 (cit. on p. 20).

[37] Sergei Gorlatch and Murray Cole. “Parallel Skeletons”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1417–1422. DOI: 10.1007/978-0-387-09766-4_24 (cit. on p. 20).

[38] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991 (cit. on p. 20).

[39] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. “SkelCL - A Portable Skeleton Library for High-Level GPU Programming”. In: *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*. 2011, pp. 1176–1182. DOI: 10.1109/IPDPS.2011.269 (cit. on p. 20).

[40] Horacio González-Vélez and Mario Leyton. “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers”. In: *Softw., Pract. Exper.* 40.12 (2010), pp. 1135–1160. DOI: 10.1002/spe.1026 (cit. on p. 21).

[41] Top500 Team. *Top500 Supercomputer List, June 2018*. Accessed: 01-07-2018 (cit. on p. 22).

[42] Bernard Gendron and Teodor Gabriel Crainic. “Parallel Branch-and-Branch Algorithms: Survey and Synthesis”. In: *Operations Research* 42.6 (1994), pp. 1042–1066. DOI: 10.1287/opre.42.6.1042 (cit. on pp. 22, 23, 25).

[43] Jan Gmys et al. “Work stealing with private integer-vector-matrix data structure for multi-core branch-and-bound algorithms”. In: *Concurrency and Computation: Practice and Experience* 28.18 (2016), pp. 4463–4484. DOI: 10.1002/cpe.3771 (cit. on pp. 22, 35).

[44] Jan Gmys et al. “A GPU-based Branch-and-Bound algorithm using Integer-Vector-Matrix data structure”. In: *Parallel Computing* 59 (2016), pp. 119–139. DOI: 10.1016/j.parco.2016.01.008 (cit. on pp. 22, 35).

[45] Ahcène Bendjoudi, Nouredine Melab, and El-Ghazali Talbi. “FTH-B&B: A Fault-Tolerant HierarchicalBranch and Bound for Large ScaleUnreliable Environments”. In: *IEEE Trans. Computers* 63.9 (2014), pp. 2302–2315. DOI: 10.1109/TC.2013.40 (cit. on pp. 22, 37).

[46] Lars Otten and Rina Dechter. “AND/OR Branch-and-Bound on a Computational Grid”. In: *J. Artif. Intell. Res.* 59 (2017), pp. 351–435. DOI: 10.1613/jair.5456 (cit. on pp. 23, 27, 76).

[47] Harry W.J.M. Trienekens. “Parallel Branch and Bound Algorithms”. PhD thesis. Erasmus University Rotterdam, 1990 (cit. on pp. 25, 41, 159, 163).

[48] Guo-Jie Li and B. Wah. “Coping with Anomalies in Parallel Branch-and-Bound Algorithms”. In: *Computers, IEEE Transactions on* C-35.6 (1986), pp. 568–573. ISSN: 0018-9340. DOI: 10.1109/TC.1986.5009434 (cit. on pp. 25, 41, 159, 163).

[49] Ten-Hwang Lai and Sartaj Sahni. “Anomalies in Parallel Branch-and-Bound Algorithms”. In: *International Conference on Parallel Processing, ICPP’83, Columbus, Ohio, USA, August 1983*. 1983, pp. 183–190 (cit. on pp. 25, 41, 159, 163).

[50] A. de Bruin, G.A.P. Kindervater, and H.W.J.M. Trienekens. “Asynchronous parallel branch and bound and anomalies”. In: *Parallel Algorithms for Irregularly Structured Problems*. Ed. by Afonso Ferreira and José Rolim. Vol. 980. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 363–377. DOI: 10.1007/3-540-60321-2_29 (cit. on pp. 25, 41, 159, 163).

[51] Harry WJM Trienekens and A de Bruin. *Towards a taxonomy of parallel branch and bound algorithms*. Tech. rep. Erasmus School of Economics (ESE), 1992 (cit. on p. 25).

[52] Ian P. Gent et al. “A Review of Literature on Parallel Constraint Solving”. In: *CoRR* abs/1803.10981 (2018) (cit. on p. 26).

[53] Tarek Menouer and Bertrand Le Cun. “A Parallelization Mixing OR-Tools/Gecode Solvers on Top of the Bobpp Framework”. In: *Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2013, Compiegne, France, October 28-30, 2013*. 2013, pp. 242–246. DOI: 10.1109/3PGCIC.2013.42 (cit. on p. 26).

[54] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. “Embarrassingly Parallel Search”. In: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*. 2013, pp. 596–610. DOI: 10.1007/978-3-642-40627-0_45 (cit. on pp. 26, 27, 38, 76).

[55] Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. “Embarrassingly Parallel Search in Constraint Programming”. In: *J. Artif. Intell. Res.* 57 (2016), pp. 421–464 (cit. on p. 26).

[56] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. “Improvement of the Embarrassingly Parallel Search for Data Centers”. In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. 2014, pp. 622–635. DOI: 10.1007/978-3-319-10428-7_45 (cit. on pp. 27, 28, 37, 168).

[57] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. “Scalable maximum clique computation using MapReduce”. In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 2013, pp. 74–85. DOI: 10.1109/ICDE.2013.6544815 (cit. on pp. 27, 28, 37).

[58] Amr Elmasry, Ayman Khalafallah, and Moustafa Meshry. “A scalable maximum-clique algorithm using Apache Spark”. In: *13th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2016, Agadir, Morocco, November*

29 - December 2, 2016. 2016, pp. 1–8. DOI: 10.1109/AICCSA.2016.7945631 (cit. on pp. 27, 37).

[59] Osama Talaat Ibrahim and Ahmed El-Mahdy. "An Efficient Load Balancing Method for Tree Algorithms". In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), Toulouse, France, July 18-21, 2016*. 2016, pp. 589–596. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0100 (cit. on p. 27).

[60] Vladimir Voloshinov, Sergey Smirnov, and Oleg Sukhoroslov. "Implementation and Use of Coarse-grained Parallel Branch-and-bound in Everest Distributed Environment". In: *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*. 2017, pp. 1532–1541. DOI: 10.1016/j.procs.2017.05.207 (cit. on pp. 28, 37).

[61] Yuchao Pan et al. "cOSPREY: A Cloud-Based Distributed Algorithm for Large-Scale Computational Protein Design". In: *Journal of Computational Biology* 23.9 (2016), pp. 737–749. DOI: 10.1089/cmb.2015.0234 (cit. on p. 28).

[62] Marijn J. H. Heule, Oliver Kullmann, and Armin Biere. "Cube-and-Conquer for Satisfiability". In: *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 31–59. DOI: 10.1007/978-3-319-63516-3_2 (cit. on pp. 28, 37).

[63] Marijn Heule and Hans van Maaren. "Look-Ahead Based SAT Solvers". In: *Handbook of Satisfiability*. IOS Press, 2009, pp. 155–184. DOI: 10.3233/978-1-58603-929-5-155 (cit. on p. 28).

[64] Matteo Fischetti, Michele Monaci, and Domenico Salvagnin. "Self-splitting of Workload in Parallel Computation". In: *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*. 2014, pp. 394–404. DOI: 10.1007/978-3-319-07046-9_28 (cit. on pp. 28, 37).

[65] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. "Solving Very Hard Problems: Cube-and-Conquer, a Hybrid SAT Solving Method". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. 2017, pp. 4864–4868. DOI: 10.24963/ijcai.2017/683 (cit. on p. 28).

[66] Richard M. Karp and Yanjun Zhang. "Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation". In: *J. ACM* 40.3 (1993), pp. 765–789. DOI: 10.1145/174130.174145 (cit. on p. 29).

[67] Andrea Pietracaprina et al. “Space-Efficient Parallel Algorithms for Combinatorial Search Problems”. In: *CoRR* abs/1306.2552 (2013) (cit. on pp. 29, 71).

[68] Peter Sanders. “Randomized Receiver Initiated Load-balancing Algorithms for Tree-shaped Computations”. In: *Comput. J.* 45.5 (2002), pp. 561–573. DOI: 10.1093/comjnl/45.5.561 (cit. on p. 29).

[69] George Ostrouchov. “Parallel computing on a hypercube: An overview of the architecture and some applications”. In: *Computer Science and Statistics, Proceedings of the 19th Symposium on the Interface*. January 1987, pp. 27–32 (cit. on p. 29).

[70] George Karypis and Vipin Kumar. “Unstructured Tree Search on SIMD Parallel Computers”. In: *IEEE Trans. Parallel Distrib. Syst.* 5.10 (1994), pp. 1057–1072. DOI: 10.1109/71.313122 (cit. on pp. 29, 33, 37).

[71] Mihai Budiu, Daniel Delling, and Renato Fonseca F. Werneck. “DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines”. In: *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. 2011, pp. 1278–1289. DOI: 10.1109/IPDPS.2011.121 (cit. on pp. 30, 37).

[72] Michael Isard et al. “Dryad: distributed data-parallel programs from sequential building blocks”. In: *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*. 2007, pp. 59–72. DOI: 10.1145/1272996.1273005 (cit. on p. 30).

[73] David Avis and Charles Jordan. “mts: a light framework for parallelizing tree search codes”. In: *CoRR* abs/1709.07605 (2017). arXiv: 1709.07605 (cit. on pp. 30, 37, 83).

[74] Mohamed Benaïchouche et al. “Building a parallel branch and bound library”. In: *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques*. 1996, pp. 201–231. DOI: 10.1007/BFb0027123 (cit. on pp. 30, 37, 38, 69).

[75] François Galea and Bertrand Le Cun. “Bob++: a framework for exact combinatorial optimization methods on parallel machines”. In: *International Conference High Performance Computing & Simulation*. 2007, pp. 779–785 (cit. on pp. 30, 37, 38, 62, 69).

[76] Tarek Menouer. “Solving combinatorial problems using a parallel framework”. In: *J. Parallel Distrib. Comput.* 112 (2018), pp. 140–153. DOI: 10.1016/j.jpdc.2017.05.019 (cit. on pp. 30, 31, 37, 38, 69).

[77] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. “KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors”. In: *Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada*. 2007, pp. 15–23. DOI: 10.1145/1278177.1278182 (cit. on p. 31).

[78] Ricardo C. Corrêa. "A Parallel Formulatiion for General Branch-and-Bound Algorithms". In: *Parallel Algorithms for Irregularly Structured Problems, Second International Workshop, IRREGULAR '95, Lyon, France, September 4-6, 1995, Proceedings*. 1995, pp. 395–409. DOI: 10.1007/3-540-60321-2_31 (cit. on p. 31).

[79] Juan F. R. Herrera et al. "On parallel Branch and Bound frameworks for Global Optimization". In: *Journal of Global Optimization* (2017), pp. 1–14. ISSN: 1573-2916. DOI: 10.1007/s10898-017-0508-y (cit. on pp. 31, 38).

[80] Gara Miranda-Valladares and Coromoto León. "OpenMP Skeletons for Tree Searches". In: *14th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2006), 15-17 February 2006, Montbeliard-Sochaux, France*. 2006, pp. 423–430. DOI: 10.1109/PDP.2006.52 (cit. on p. 31).

[81] Michael Poldner and Herbert Kuchen. "Algorithmic skeletons for branch & bound". In: *ICSOFT 2006, First International Conference on Software and Data Technologies, Setúbal, Portugal, September 11-14, 2006*. 2006, pp. 291–300. DOI: 10.1007/978-3-540-70621-2_17 (cit. on pp. 31, 33, 37, 38).

[82] Enrique Alba et al. "MALLBA: A Library of Skeletons for Combinatorial Optimisation (Research Note)". In: *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27-30, 2002, Proceedings*. 2002, pp. 927–932. DOI: 10.1007/3-540-45706-2_132 (cit. on pp. 31, 38).

[83] Isabel Dorta, Coromoto León, and Casiano Rodríguez. "Performance analysis of Branch-and-Bound skeletons". In: *Mathematical and Computer Modelling* 51.3-4 (2010), pp. 300–308. DOI: 10.1016/j.mcm.2009.08.003 (cit. on p. 31).

[84] Christian Schulte. "Parallel search made simple". In: *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP*. 2000, pp. 41–57 (cit. on pp. 31, 37).

[85] Adel Dabah, Ahcène Bendjoudi, and Abdelhakim AitZai. "Efficient parallel B&B method for the blocking job shop scheduling problem". In: *International Conference on High Performance Computing & Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016*. 2016, pp. 784–791. DOI: 10.1109/HPCSim.2016.7568414 (cit. on pp. 31, 32, 37, 75).

[86] Jonathan Eckstein, Cynthia A. Phillips, and William E. Hart. "Pico: An Object-Oriented Framework for Parallel Branch and Bound". In: *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*. Ed. by Yair Censor Dan Butnariu and Simeon Reich. Vol. 8. Studies in Computational Mathematics. Elsevier, 2001, pp. 219–265. DOI: 10.1016/S1570-579X(01)80014-8 (cit. on pp. 31, 38, 85).

[87] Jonathan Eckstein, William E. Hart, and Cynthia A. Phillips. "PEBBL: an object-oriented framework for scalable parallel branch and bound". In: *Math. Program. Comput.* 7.4 (2015), pp. 429–469. DOI: 10.1007/s12532-015-0087-1 (cit. on pp. 31, 37, 38).

[88] Yan Xu et al. "Alps: A Framework for Implementing Parallel Tree Search Algorithms". In: *The Next Wave in Computing, Optimization, and Decision Technologies*. Boston, MA: Springer US, 2005, pp. 319–334. DOI: 10.1007/0-387-23529-9_21 (cit. on pp. 32, 37, 38).

[89] Joxan Jaffar et al. "Scalable Distributed Depth-First Search with Greedy Work Stealing". In: *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*. 2004, pp. 98–103. DOI: 10.1109/ICTAI.2004.107 (cit. on pp. 32, 37).

[90] Trong-Tuan Vu et al. "Overlay-Centric Load Balancing: Applications to UTS and B&B". In: *2012 IEEE International Conference on Cluster Computing, CLUSTER 2012, Beijing, China, September 24-28, 2012*. 2012, pp. 382–390. DOI: 10.1109/CLUSTER.2012.17 (cit. on pp. 32, 37).

[91] Ahcène Bendjoudi, Nouredine Melab, and El-Ghazali Talbi. "Fault-Tolerant Mechanism for Hierarchical Branch and Bound Algorithm". In: *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*. 2011, pp. 1806–1814. DOI: 10.1109/IPDPS.2011.339 (cit. on p. 32).

[92] Raphael A. Finkel and Udi Manber. "DIB - A Distributed Implementation of Backtracking". In: *ACM Trans. Program. Lang. Syst.* 9.2 (1987), pp. 235–256. DOI: 10.1145/22719.24067 (cit. on pp. 32, 36, 37).

[93] Peter Sanders. "Better Algorithms for Parallel Backtracking". In: *Parallel Algorithms for Irregularly Structured Problems, Second International Workshop, IRREGULAR '95, Lyon, France, September 4-6, 1995, Proceedings*. 1995, pp. 333–347. DOI: 10.1007/3-540-60321-2_27 (cit. on p. 33).

[94] Alexander Reinefeld. "Parallel search in discrete optimization problems". In: *Simul. Pr. Theory* 4.2-3 (1996), pp. 169–188. DOI: 10.1016/0928-4869(95)00036-4 (cit. on p. 33).

[95] Yanjun Zhang and A. Ortynski. "Efficiency of Randomized Parallel Backtrack Search". In: *Algorithmica* 24.1 (1999), pp. 14–28. DOI: 10.1007/PL00009269 (cit. on p. 33).

[96] Trong-Tuan Vu and Bilel Derbel. "Parallel Branch-and-Bound in multi-core multi-CPU multi-GPU heterogeneous environments". In: *Future Generation Comp. Syst.* 56 (2016), pp. 95–109. DOI: 10.1016/j.future.2015.10.009 (cit. on pp. 33, 37).

[97] Adrian Brüngger et al. “The parallel search bench ZRAM and its applications”. In: *Annals OR* 90 (1999), pp. 45–63. DOI: 10.1023/A3A1018972901171 (cit. on pp. 33, 37).

[98] Reinhard Lüling et al. “Mapping tree-structured combinatorial optimization problems onto parallel computers”. In: *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques*. 1996, pp. 115–144. DOI: 10.1007/BFb0027120 (cit. on pp. 34, 37).

[99] Chao-Yang Gau and Mark A. Stadtherr. “Parallel Branch-and-Bound for Chemical Engineering Applications: Load Balancing and Scheduling Issues”. In: *Vector and Parallel Processing - VECPAR 2000, 4th International Conference, Porto, Portugal, June 21-23, 2000, Selected Papers and Invited Talks*. 2000, pp. 273–300. DOI: 10.1007/3-540-44942-6_24 (cit. on pp. 34, 37).

[100] Giuseppe Di Fatta and Michael R. Berthold. “Decentralized Load Balancing for Highly Irregular Search Problems”. In: *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC 2006), 26-29 June 2006, Cagliari, Sardinia, Italy*. 2006, pp. 220–226. DOI: 10.1109/ISCC.2006.56 (cit. on pp. 34, 37).

[101] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. “Confidence-Based Work Stealing in Parallel Constraint Programming”. In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*. 2009, pp. 226–241. DOI: 10.1007/978-3-642-04244-7_20 (cit. on pp. 34, 37, 38, 72, 78, 83).

[102] Faisal N. Abu-Khzam et al. “On scalable parallel recursive backtracking”. In: *Journal of Parallel and Distributed Computing* 84 (2015), pp. 65–75. DOI: 10.1016/j.jpdc.2015.07.006 (cit. on pp. 34, 36, 37, 70, 78).

[103] Mohand Mezmaz et al. “A Multi-core Parallel Branch-and-Bound Algorithm Using Factorial Number System”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. 2014, pp. 1203–1212. DOI: 10.1109/IPDPS.2014.124 (cit. on pp. 35, 37).

[104] J. Falcou et al. “Quaff: efficient C++ design for parallel skeletons”. In: *Parallel Computing* 32.7–8 (2006). Algorithmic Skeletons, pp. 604–615. ISSN: 0167-8191. DOI: 10.1016/j.parco.2006.06.001 (cit. on pp. 38, 66, 97).

[105] Blair Archibald. *YewPar Source Repository*. Thesis release. DOI: 10.5281/zenodo.1316476 (cit. on pp. 39, 118).

[106] Gordon D. Plotkin. “The origins of structural operational semantics”. In: *J. Log. Algebr. Program.* 60-61 (2004), pp. 3–15. DOI: 10.1016/j.jlap.2004.03.009 (cit. on p. 41).

[107] Christian Schulte and Guido Tack. “Weakly Monotonic Propagators”. In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*. 2009, pp. 723–730. DOI: 10.1007/978-3-642-04244-7_56 (cit. on p. 55).

[108] Isabel Dorta et al. “Parallel Skeletons for Divide-and-Conquer and Branch-and-Bound techniques”. In: *11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2003), 5-7 February 2003, Genova, Italy*. 2003, pp. 292–298. DOI: 10.1109/EMPDP.2003.1183602 (cit. on p. 62).

[109] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. *The Münster Skeleton Library Muesli: A comprehensive overview*. Tech. rep. Working Papers, ERCIS-European Research Center for Information Systems, 2009 (cit. on p. 62).

[110] John Darlington et al. “Parallel Programming Using Skeleton Functions”. In: *PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Munich, Germany, June 14-17, 1993, Proceedings*. 1993, pp. 146–160. DOI: 10.1007/3-540-56891-3_12 (cit. on p. 67).

[111] Philippe Charles et al. “X10: an object-oriented approach to non-uniform cluster computing”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. 2005, pp. 519–538. DOI: 10.1145/1094811.1094852 (cit. on pp. 69, 96).

[112] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. “Parallel Programmability and the Chapel Language”. In: *IJHPCA* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442 (cit. on pp. 69, 96).

[113] Christian Schulte. “Comparing Trailing and Copying for Constraint Programming”. In: *Proceedings of the Sixteenth International Conference on Logic Programming*. Ed. by Danny De Schreye. Las Cruces, NM, USA: The MIT Press, November 1999, pp. 275–289 (cit. on p. 70).

[114] Imen Chakroun and Nouredine Melab. “HB&B@GRID: An heterogeneous grid-enabled Branch and Bound algorithm”. In: *International Conference on High Performance Computing & Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016*. 2016, pp. 697–704. DOI: 10.1109/HPCSim.2016.7568403 (cit. on p. 75).

[115] Shintaro Iwasaki and Kenjiro Taura. “Autotuning of a Cut-Off for Task Parallel Programs”. In: *10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSOC 2016, Lyon, France, September 21-23, 2016*. 2016, pp. 353–360. DOI: 10.1109/MCSoC.2016.51 (cit. on p. 76).

[116] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The Implementation of the Cilk-5 Multithreaded Language”. In: *Proceedings of the ACM SIGPLAN '98*

Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998. 1998, pp. 212–223. DOI: 10.1145/277650.277725 (cit. on p. 85).

[117] Ciaran McCreesh and Patrick Prosser. "The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound". In: *TOPC* 2.1 (2015), 8:1–8:27. DOI: 10.1145/2742359 (cit. on p. 86).

[118] Jean Fromentin and Florent Hivert. "Exploring the tree of numerical semigroups". In: *Math. Comp.* 85.301 (2016), pp. 2553–2568. ISSN: 0025-5718. DOI: 10.1090/mcom/3075 (cit. on pp. 88, 108).

[119] Patrick Maier, Robert Stewart, and Phil Trinder. "The HdpH DSLs for Scalable Reliable Computation". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: ACM, 2014, pp. 65–76. DOI: 10.1145/2633357.2633363 (cit. on p. 94).

[120] Thomas Heller et al. "HPX – An open source C++ Standard Library for Parallelism and Concurrency". In: *Proceedings of OpenSuCo 2017, Denver, Colorado USA, November 2017 (OpenSuCo'17)*. 2017, p. 5 (cit. on pp. 95, 188).

[121] Thorsten von Eicken et al. "Active Messages: A Mechanism for Integrated Communication and Computation". In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. 1998, pp. 430–440. DOI: 10.1145/285930.286002 (cit. on p. 96).

[122] Robert H. Halstead Jr. "Multilisp: A Language for Concurrent Symbolic Computation". In: *ACM Trans. Program. Lang. Syst.* 7.4 (1985), pp. 501–538. DOI: 10.1145/4472.4478 (cit. on p. 96).

[123] Vivek Kumar et al. "HabaneroUPC++: a Compiler-free PGAS Library". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, OR, USA, October 6-10, 2014*. 2014, 5:1–5:10. DOI: 10.1145/2676870.2676879 (cit. on p. 96).

[124] Stephen Olivier et al. "UTS: An unbalanced tree search benchmark". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2006, pp. 235–250 (cit. on p. 106).

[125] Vijay A. Saraswat et al. "Lifeline-based global load balancing". In: *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*. 2011, pp. 201–212. DOI: 10.1145/1941553.1941582 (cit. on pp. 106, 187).

[126] Vivek Kumar et al. "Optimized Distributed Work-Stealing". In: *6th Workshop on Irregular Applications: Architecture and Algorithms, IA3@SC 2016, Salt Lake City,*

UT, USA, November 13, 2016. 2016, pp. 74–77. DOI: 10.1109/IA3.2016.019 (cit. on pp. 106, 187).

[127] Donald E. Eastlake III and Paul E. Jones. "US Secure Hash Algorithm 1 (SHA1)". In: *RFC* 3174 (2001), pp. 1–22. DOI: 10.17487/RFC3174 (cit. on p. 106).

[128] UTS Project Members. *The Unbalanced Tree Search Benchmark*. https://sourceforge.net/p/uts-benchmark/wiki/Home/. 2018 (cit. on p. 106).

[129] Jonas Posner and Claudia Fohry. "Cooperation vs. coordination for lifeline-based global load balancing in APGAS". In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10, X10@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*. 2016, pp. 13–17. DOI: 10.1145/2931028.2931029 (cit. on p. 106).

[130] Max Grossman et al. "A Pluggable Framework for Composable HPC Scheduling Libraries". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*. 2017, pp. 723–732. DOI: 10.1109/IPDPSW.2017.13 (cit. on p. 106).

[131] M. Bras-Amorós and J. Fernández-González. "Computation of numerical semigroups by means of seeds". In: *ArXiv e-prints* abs/1411.6093 (July 2016). arXiv: 1607.01545 [math.CO] (cit. on p. 107).

[132] A. Assi and P. A. García-Sánchez. "Numerical semigroups and applications". In: *ArXiv e-prints* abs/1411.6093 (November 2014). arXiv: 1411.6093 [math.AG] (cit. on p. 107).

[133] Qinghua Wu and Jin-Kao Hao. "A review on algorithms for maximum clique problems". In: *European Journal of Operational Research* 242.3 (2015), pp. 693–709. DOI: 10.1016/j.ejor.2014.09.064 (cit. on p. 109).

[134] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. "An exact bit-parallel algorithm for the maximum clique problem". In: *Computers & OR* 38.2 (2011), pp. 571–581. DOI: 10.1016/j.cor.2010.07.019 (cit. on p. 109).

[135] Pablo San Segundo et al. "An improved bit parallel exact maximum clique algorithm". In: *Optimization Letters* 7.3 (2013), pp. 467–479. DOI: 10.1007/s11590-011-0431-y (cit. on p. 109).

[136] C. W. H. Lam. "The Search for a Finite Projective Plane of Order 10". In: *The American Mathematical Monthly* 98.4 (1991), pp. 305–318. ISSN: 00029890, 19300972 (cit. on p. 110).

[137] Andreas Klein and Leo Storme. "Applications of finite geometry in coding theory and cryptography". eng. In: *NATO Science for Peace and Security, Series D : Information and Communication Security*. Ed. by Dean Crnković and Vladimir Tonchev. Vol. 29. Opatija, Croatia: IOS Press, 2011, pp. 38–58. DOI: 10.3233/978-1-60750-663-8-38 (cit. on p. 110).

[138] *GAP – Groups, Algorithms, and Programming, Version 4.8.7*. The GAP Group. 2017 (cit. on p. 111).

[139] John Bamberg et al. *FinInG – Finite Incidence Geometry, Version 1.3*. 2015 (cit. on p. 111).

[140] Leonard Soicher. *The GRAPE package for GAP, Version 4.7*. 2016 (cit. on p. 111).

[141] Toby Walsh. "General Symmetry Breaking Constraints". In: *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*. 2006, pp. 650–664. DOI: 10.1007/11889205_46 (cit. on p. 111).

[142] James Ostrowski et al. "Orbital branching". In: *Math. Program.* 126.1 (2011), pp. 147–178. DOI: 10.1007/s10107-009-0273-x (cit. on p. 111).

[143] Ciaran McCreesh and Patrick Prosser. "A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs". In: *Principles and Practice of Constraint Programming*. Ed. by Gilles Pesant. Cham: Springer International Publishing, 2015, pp. 295–312 (cit. on p. 112).

[144] Christine Solnon. *Benchmarks for the Subgraph Isomorphism Problem*. Accessed: 10-04-2018 (cit. on pp. 112, 113).

[145] David J. Johnson and Michael A. Trick, eds. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. Boston, MA, USA: American Mathematical Society, 1996 (cit. on pp. 113, 119).

[146] Rajesh Matai, Surya Singh, and Murari Lal Mittal. "Traveling salesman problem: an overview of applications, formulations, and solution approaches". In: *Traveling Salesman Problem, Theory and Applications*. InTech, 2010 (cit. on p. 114).

[147] Concorde Developers. *Concorde Site*. Accessed: 22-03-2018 (cit. on p. 114).

[148] R. C. Prim. "Shortest connection networks and some generalizations". In: *The Bell System Technical Journal* 36.6 (November 1957), pp. 1389–1401. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1957.tb01515.x (cit. on p. 114).

[149] David Johnson et al. *Eighth DIMACS Implementation Challenge*. Accessed: 05-12-2016 (cit. on p. 114).

[150] Harvey M. Salkin and Cornelis A. De Kluyver. "The knapsack problem: A survey". In: *Naval Research Logistics Quarterly* 22.1 (1975), pp. 127–144. ISSN: 1931-9193. DOI: 10.1002/nav.3800220110 (cit. on p. 115).

[151] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990 (cit. on p. 115).

[152] Silvano Martello, David Pisinger, and Paolo Toth. "New trends in exact algorithms for the 0-1 knapsack problem". In: *European Journal of Operational Research* 123.2 (2000), pp. 325–332. DOI: 10.1016/S0377-2217(99)00260-X (cit. on p. 115).

[153] Jiří Matoušek and Bernd Gärtner. *Integer Programming and LP Relaxation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 29–40. DOI: 10.1007/978-3-540-30717-4_3 (cit. on p. 115).

[154] David Pisinger. "Where are the hard knapsack problems?" In: *Computers & Operations Research* 32.9 (2005), pp. 2271–2284. ISSN: 0305-0548. DOI: 10.1016/j.cor.2004.03.002 (cit. on pp. 115, 116).

[155] David Pisinger. *David Pisinger's optimization codes – hard knapsack instances*. Accessed: 14-10-2016 (cit. on pp. 115, 116).

[156] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. "Nix: A Safe and Policy-Free System for Software Deployment". In: *Proceedings of the 18th Conference on Systems Administration (LISA 2004), Atlanta, USA, November 14-19, 2004*. 2004, pp. 79–92 (cit. on p. 118).

[157] Blair Archibald. *Algorithmic Skeletons for Exact Combinatorial Search at Scale [Data Collection]*. 2018. DOI: 10.5525/gla.researchdata.644 (cit. on p. 118).

[158] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560 (cit. on p. 118).

[159] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (1988), pp. 532–533. DOI: 10.1145/42411.42415 (cit. on p. 118).

[160] Ciaran McCreesh. *Sequential MCsa1 Maximum Clique Implementation*. Accessed: 04-07-2018 (cit. on p. 119).

[161] Matthew W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. 2001, pp. 530–535. DOI: 10.1145/378239.379017 (cit. on p. 121).

[162] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. *Modeling and programming with Gecode*. Version 5.0.0. 2016 (cit. on p. 187).

[163] André A. Ciré, Serdar Kadioglu, and Meinolf Sellmann. "Parallel Restarted Search". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.* 2014, pp. 842–848 (cit. on p. 187).

[164] Thomas Schiex and Gérard Verfaillie. "Nogood Recording for Static and Dynamic Constraint Satisfaction Problems". In: *Fifth International Conference on Tools with Artificial Intelligence, ICTAI '93, Boston, Massachusetts, USA, November 8-11, 1993*. 1993, pp. 48–55. DOI: 10.1109/TAI.1993.633935 (cit. on p. 187).

[165] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. “Sequential Model-Based Optimization for General Algorithm Configuration”. In: *Learning and Intelligent Optimization*. Ed. by Carlos A. Coello Coello. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 507–523 (cit. on p. 188).

Glossary

MT^3

A formal, operational semantics, for parallel tree searches. Chapter 3.

Budget

A parallel search coordination. Workers spawn tasks from their generator stack when a backtrack limit (the budget) is reached. Section 4.3.6.

Depth-Bounded

A parallel search coordination. Converts all search tree nodes below a cutoff depth d_{cutoff} into tasks. Section 4.3.4.

HPX

A C++ framework for asynchronous task-parallelism with support for distributed-memory architectures. Section 5.1.2.1.

Ordered

A parallel search coordination. Generates tasks upfront and explores these in a fixed sequential and parallel ordering to ensure replicable performance. Section 7.4.

PruneLevel

An optimisation where, on failing a bound check, the failed node and any node *to the right* is pruned. Section 3.4.3.

Sequential

A search coordination. Performs a completely sequential depth-first search. Section 4.3.3.

Stack-Stealing

A parallel search coordination. Requests tasks directly from running workers who returns sub-trees from their generator stack. Section 4.3.5.

Tree Search Framework (TSF)

An abstract framework for tree search, used to design the skeletons in a implementation independent manner..

YewPar

A Framework for distributed-memory parallel tree search featuring the skeletons described in this thesis. Section 5.1.2.

Appendix A

MT^3 Rule Listings

The complete set of MT^3 rules, from both Chapter 3 are Chapter 4, are given below.

Traversal Rules (Section 3.3.4)

$$\text{(advance}_i\text{)} \quad \frac{u = succ(S, v) \qquad u \neq \bot}{\langle \sigma, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \sigma, \mathit{Tasks}, \ldots, \langle S, u\rangle, \ldots\rangle}$$

$$\text{(terminate}_i\text{)} \quad \frac{succ(S, v) = \bot}{\langle \sigma, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \sigma, \mathit{Tasks}, \ldots, \bot, \ldots\rangle}$$

$$\text{(schedule}_i\text{)} \quad \frac{v = \text{root of } S}{\langle \sigma, S{:}\mathit{Tasks}, \ldots, \bot, \ldots\rangle \rightarrow \langle \sigma, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle}$$

Node Processing Rules (Section 3.3.1)

$$\text{(enumerate}_i\text{)} \quad \frac{}{\langle \{m\}_e, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \{m + h(v)\}_e, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle}$$

$$\text{(decide}_i\text{)} \quad \frac{match(v)}{\langle \{\}_d, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \{v\}_d, [\,], \bot, \ldots, \bot\rangle}$$

$$\text{(optimise}_i\text{)} \quad \frac{improves(u, v)}{\langle \{u\}_o, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle \rightarrow \langle \{v\}_o, \mathit{Tasks}, \ldots, \langle S, v\rangle, \ldots\rangle}$$

Pruning Rules (Section 3.3.7)

$$\text{(prune-decide}_i\text{)}\quad \frac{p_d(v) \qquad S' = subtree(S, v)}{\langle \{\epsilon\}_d, Tasks, \ldots, \langle S, v\rangle, \ldots\rangle \to \langle \{\epsilon\}_d, Tasks, \ldots, \langle (S \setminus S') \cup \{v\}, v\rangle, \ldots\rangle}$$

$$\text{(prune-optimise}_i\text{)}\quad \frac{p_o(u, v) \qquad S' = subtree(S, v)}{\langle \{u\}_o, Tasks, \ldots, \langle S, v\rangle, \ldots\rangle \to \langle \{u\}_o, Tasks, \ldots, \langle (S \setminus S') \cup \{v\}, v\rangle, \ldots\rangle}$$

Spawn Rules (Sections 3.3.8, 4.3.4.1, 4.3.5.1 and 4.3.6.1)

$$\text{(spawn}_i\text{)}\quad \frac{u \in S \qquad v <_{\text{lex}} u \qquad S_u = subtree(S, u)}{\langle \sigma, Tasks, \ldots, \langle S, v\rangle, \ldots\rangle \to \langle \sigma, Tasks{:}S_u, \ldots, \langle S \setminus S_u, v\rangle, \ldots\rangle}$$

$$\text{(spawn-depth-bounded}_i\text{)}\quad \frac{|v| + 1 \leq d_{cutoff} \qquad \{S_{c_1} \ldots S_{c_n}\} = \{subtree(S, u) \mid u \in children(v)\}}{\langle \sigma, Tasks, \ldots, \langle S, v\rangle, \ldots\rangle \to \langle \sigma, Tasks{:}S_{c_1} \ldots{:}S_{c_n}, \ldots, \langle S \setminus S_{c_1} \setminus \cdots \setminus S_{c_n}, v\rangle, \ldots\rangle}$$

$$\text{(spawn-stack-stealing}_i\text{)}\quad \frac{u = nextLowest(S, v) \qquad u \neq \emptyset \qquad S_u = subtree(S, u)}{\langle \sigma, [\,], \bot, \langle S, v\rangle, \ldots\rangle \to \langle \sigma, [S_u], \ldots, \langle S \setminus S_u, v\rangle, \ldots\rangle}$$

$$\text{(spawn-budget}_i\text{)}\quad \frac{backtracks(i) = budget \quad \{c_1, \ldots, c_n\} = lowest(S, v) \quad \{S_{c_1}, \ldots, S_{c_n}\} = subtree(S, u)}{\langle \sigma, Tasks, \ldots, \langle S, v\rangle, \ldots\rangle \to \langle \sigma, Tasks{:}S_{c_1}{:}\ldots{:}S_{c_n}, \ldots, \langle S \setminus S_{c_1} \setminus \cdots \setminus S_{c_n}, v\rangle, \ldots\rangle \quad backtracks(i) = 0}$$

Appendix B

Executable MT^3 Rules

The following Haskell program implements the branch and bound optimisation variant of MT^3 to show the reductions can be implemented. It does not support the spawning of new tasks, although it could be extended to do so. The application can be tested by running the `runExample` function within `ghci`.

```
{-# LANGUAGE ViewPatterns #-}
{-# LANGUAGE OverloadedStrings #-}

{-
   Simple encoding of the branch and bound optimisation rules of MT^3
   Spawning is not supported, assumes initial task set contains required tasks
-}

import qualified Data.Set as S
import qualified Data.Map.Strict as M
import Data.List

-- NodeLabel * Objective Value * Bound
data Node = Node [Int] Int Int deriving (Show)

instance Eq Node where
  Node x _ _ == Node y _ _ = x == y

instance Ord Node where
  Node x _ _ `compare` Node y _ _ = x `compare` y

isPrefixOf_ :: Node -> Node -> Bool
isPrefixOf_ (Node n1 _ _) (Node n2 _ _) = n1 `isPrefixOf` n2

obj :: Node -> Int
obj (Node _ o _) = o

bnd :: Node -> Int
bnd (Node _ _ b) = b

-- (Sub) Trees are sets of nodes
type SubTree = S.Set Node

-- A subtree rooted at N is all Nodes in S with a common prefix of N (including N)
```

```
removeChildren :: SubTree -> Node -> (SubTree, SubTree)
removeChildren s n =  S.partition (n `isPrefixOf_`) s

succ_ :: SubTree -> Node -> Maybe Node
succ_ s n = let future = snd $ S.partition (<= n) s
            in if null future then Nothing else Just $ S.findMin future

-- Thread States
type ThreadId = Int
data RuleCategory = Null | Traversal | NodeProcessing | Pruning deriving (Show, Eq)

data ThreadState =
    ThreadState {subtree :: SubTree , view :: Node, lastRuleType :: RuleCategory}
  | Bot deriving (Show, Eq)

threadFromSubTree :: SubTree -> ThreadState
threadFromSubTree s = ThreadState s (S.findMin s) Traversal

-- Search State (sigma)
data State = State {
    incumbent :: Node
  , tasks     :: [SubTree]
  , threads   :: M.Map ThreadId ThreadState
  , ruleLog :: [String]
  } deriving (Show, Eq)

isEnd :: Int -> State -> Bool
isEnd n st = tasks st == [] && threads st == M.fromList [(i,Bot) | i <- [0..n] ]

-- Rules. Return Nothing if they cannot be applied
type Rule = ThreadId -> State -> Maybe State

advance :: Rule
advance i st@(State _ _ ((M.! i) -> Bot) _) = Nothing
advance i st@(State inc _ ts@((M.! i) -> ThreadState s v _) _) =
  case succ_ s v of
    Just n -> Just st { threads = M.insert i (ThreadState s n Traversal) ts, ruleLog = (
        ruleLog st ++ ["t" ++ show i ++ " advance"]) }
    Nothing -> Nothing

terminate :: Rule
terminate i st@(State _ _ ((M.! i) -> Bot) _) = Nothing
terminate i st@(State inc _ ts@((M.! i) -> ThreadState s v _) _) =
  case succ_ s v of
    Nothing -> Just st { threads = M.insert i Bot ts, ruleLog = (ruleLog st ++ ["t" ++ show
        i ++ " terminate"])  }
    Just _  -> Nothing

schedule :: Rule
schedule i st@(State _ [] _ _) = Nothing
schedule i st@(State _ (t:tasks') ts@((M.! i) -> Bot) _) =
  Just st { tasks = tasks', threads = M.insert i (threadFromSubTree t) ts, ruleLog = (
      ruleLog st ++ ["t" ++ show i ++ " schedule"])   }

optimise :: Rule
optimise i st@(State _ _ ((M.! i) -> Bot) _) = Nothing
optimise i st@(State inc _ ts@((M.! i) -> s@(ThreadState _ v _)) _) =
  if obj v <= obj inc
```

```
      then Nothing
      else Just st { incumbent = v, threads = M.insert i (s {lastRuleType = NodeProcessing
          }) ts, ruleLog = (ruleLog st ++ ["t" ++ show i ++ " optimise"]) }

pruneopt :: Rule
pruneopt i st@(State _ _ ((M.! i) -> Bot) _) = Nothing
pruneopt i st@(State inc _ ts@((M.! i) -> ThreadState s v _) _) =
  if bnd v > obj inc
      then Nothing
      else Just st { threads = M.insert i (ThreadState (snd $ removeChildren s v) v Pruning
          ) ts, ruleLog = (ruleLog st ++ ["t" ++ show i ++ " prune-opt"])  }

-- Rule executor
untilSuc :: a -> [a -> Maybe b] -> Maybe b
untilSuc st fns = go fns
  where
    go [] = Nothing
    go (f:fs) = case f st of
                  Just x -> Just x
                  Nothing -> go fs

-- Get the next rule ordering based on the previous rule.
getRuleSet :: ThreadId -> RuleCategory -> [State -> Maybe State]
getRuleSet i Null = getRuleSet i Pruning
getRuleSet i Traversal = [optimise i, pruneopt i, advance i, schedule i, terminate i]
getRuleSet i NodeProcessing = [pruneopt i, advance i, schedule i, terminate i]
getRuleSet i Pruning = [advance i, schedule i, terminate i, optimise i, pruneopt i]

-- Could do with adding in better output here
step :: ThreadId -> State -> State
step i st@(State _ _ ts _) =
  case ts M.! i of
    Bot -> step' Traversal
    ThreadState _ _ last -> step' last
  where
    step' l = let rules = getRuleSet i l
              in case untilSuc st rules of
                  Just s -> s
                  Nothing -> st

-- -- Reduce the subtrees on N threads
reduce :: Int -> [SubTree] -> State
reduce numTs trees =
  let initSt = State (Node [] 0 0) trees (M.fromList [(i,Bot) | i <- [0..numTs - 1]]) []
  in until (isEnd $ numTs - 1) stepFn initSt
  where
    stepFn s = foldl foldFn s [0..numTs - 1]
    foldFn st' n = if isEnd (numTs - 1) st' then st' else step n st'

-- Example search tree from Chapter 7
exampleTree :: SubTree
exampleTree =
  S.fromList [Node [0] 0 35,
                Node [0,0] 0 30,
                Node [0,0,0] 0 22,
                  Node [0,0,0,0] 14 14,
                  Node [0,0,0,1] 16 16,
                Node [0,0,1] 0 21,
```

```
                Node [0,0,1,0] 15 15,
              Node [0,1] 0 29,
              Node [0,1,0] 0 23,
                Node [0,1,0,0] 20 20,
                Node [0,1,0,1] 16 16,
              Node [0,2] 0 29,
              Node [0,2,0] 0 12,
                Node [0,2,0,0] 11 11,
              Node [0,2,1] 0 5,
                Node [0,2,1,0] 2 2,
              Node [0,3] 0 18,
              Node [0,3,0] 0 12,
                Node [0,3,0,0] 8 8,
              Node [0,3,1] 0 11,
                Node [0,3,1,0] 7 7,
                Node [0,3,1,1] 3 3
            ]

runExample = reduce 1 [exampleTree]
```

Appendix C

Repeatability of Scaling

The thrid property of replicable branch and bound search (Chapter 7) is repeatability. Repeatability states that: "Parallel runtimes of repeated searches on the same parallel configuration have low variance.", but what do we mean by low variance?

We define the repeatability of measurements using relative standard deviation (RSD) that is defined as $\frac{std(x)}{mean(x)} \times 100$. Repeatability is a relative term, that is, we can say 10% repeatability is better than 20%. However, there is no predetermined RSD percentages that represents *good* repeatability as this depends on the context.

In this appendix we derive appropriate values of *good* repeatability based on RSD values required to detect scaling in parallel environments. This is only one of many possible measures of *good* repeatability. For example, if we were studying the effect of algorithmic changes in an identical parallel environment then we may need lower values of RSD to detect changes.

The following derivation was originally derived by Patrick Maier.

C.1 Derivation

We want to determine a suitable value of repeatability such that we can detect parallel scaling if it is present. We assume two experimental setups, one with p workers and another with q, where $p < q$.

We assume:

1. Parallel runtimes for p workers are normally distributed with mean μ_p and standard deviation σ_p.

2. $\mu_p < \mu_q$, that is, search actually scales.

3. $\frac{\sigma_p}{\mu_p} = \beta\frac{\sigma_q}{\mu_q}$. That is, relative standard deviation increases by a factor β as we increase the number of workers.

To show scalability we require the confidence intervals, e.g. $\langle \mu_p - z\sigma_p, \mu_p + z\sigma_p \rangle$, for p and q workers to be non-overlapping. Where z is based on the required confidence, e.g. setting z = 1.96 gives the 95% confidence interval.

As $p < q$ (by definition) we require:

$$\mu_q + z\sigma_q < \mu_p - z\sigma_p$$

In order to detect scaling.

Letting $\alpha = \frac{\mu_p}{\mu_q}$, we obtain:

$$\begin{aligned} z\sigma_q + z\beta\alpha\sigma_q &< \alpha\mu_q - \mu_q \\ (\beta\alpha + 1)z\sigma_q &< (\alpha - 1)\mu_q \\ \frac{\sigma_q}{\mu_q} &< \frac{\alpha - 1}{z(\beta\alpha + 1)} \end{aligned}$$

We can use this to determine the value of RSD (repeatability) required to determine scaling.

C.2 Example Values of Repeatability

Table C.1 shows the RSD values required to detect scaling with different values of α, the amount of scaling we want to observe, and β, the expected increase of RSD as we increase scale. We assume a 95% confidence in all cases.

α	β	RSD (%)	Note
3	1	17	Superlinear Scaling
2	1	17	Perfect Linear Scaling
1.8	1	14	80% Efficient Scaling
1.5	1	10	50% Efficient Scaling
2	1.1	16	10% increase in RSD at scale
2	1.5	12	50% increase in RSD at scale
1.8	1.1	13	
1.5	1.1	10	
1.5	1.5	7	

Table C.1: Example RSD values required for scaling repeatability. RSD is rounded up to the nearest percentage.

The results in Section 7.5.2 suggest that for $p \neq 1$ the differences in RSD values is generally small, e.g. $\beta = 1.1$. The scaling results in Section 7.5.1 show Ordered to scale poorly in general, e.g., $\alpha = 1.5$. Taken together, these results suggest that an RSD value of around 10% should be considered to have good repeatability, *if* we are trying to show scaling.

Appendix D

Discrepancy Ordering

The Ordered skeletons of Chapter 7 allow the parallel workers to explore the search tree in any *fixed* ordering, while still providing replicable performance guarantees. YewPar currently supports two orderings (described in Section 7.4.1): Linear, where the parallel workers explore in the same order as a sequential search, and Discrepancy, where tasks are assigned priorities based on the number of discrepancies taken to get to the starting node.

This appendix explores the effect on performance of the different orders using the benchmarks and instances of Section 7.5.

Figure D.1 compares the median runtime (over 30 runs) for linear and discrepancy ordering. As expected, for a single worker the runtimes are equal as both workers follow the sequential order. As we increase the number of workers the discrepancy search ordering performs worse than the linear ordering for almost all instances. Given the magnitude of the overhead, this is likely caused by increased management cost for the global priority queue of tasks in the discrepancy case.

Both the linear and discrepancy orders are heavily left biased (i.e. often follow the heuristic as much as possible). It is possible the the limited difference in performance is caused by many of the instances having a strong heuristic causing both orderings to find an improved bound/solution quickly. In practice additional orderings are possible and with these we might expect to observe larger performance differences.

(a) Maximum Clique

(b) Travelling Salesperson

(c) Subgraph Isomorphism

Figure D.1: Ordered Skeleton: Linear vs. Discrepancy Ordering